Separating Use and Reuse to Improve Both

Hrshikesh Arora [1]
Marco Servetto [1]
Bruno C. d. S. Oliveira [2]

[1]Victoria University of Wellington

[2]The University of Hong Kong

Subtyping and Subclassing

- Subtyping without subclassing; easy: Java interfaces
- Subclassing without subtyping; hard:

```
class A{ int ma(){ return Utils.m(this); } }
class Utils{ static int m(A a){..} }
class B extends A{ int mb(){return this.ma();} }
..
new B().mb();
```

This-leaking problem: **this** written in class **A** is of type **A**. when **B** inherits the code of **A**, code **Uitls**.m(**this**) would pass a **this** as an **A**. To be sound, **B** $\leq$ **A** must hold.

Reusing code without inducing subtyping

Three different approaches separating subtyping and subclassing:

- DeepFJig
- TraitRecordJ
- PackageTemplate

Widely different approaches, similar core ideas that we synthesize and improve: *separating code from use and code for reuse*

42_μ: our approach:

We synthesize the best from these 3 approaches.
Traits! (inspired from the original approach, not Scala/Rust traits).

- **Traits** for Code Reuse: traits can be reused to produce more code.
- **Classes** for Code Use: classes can be instantiated.
- **Types**: classes and interface names can be used as a types.
- **Exact types**: all classes are final; inheritance by reusing traits.
- **Sub types**: interfaces are the only way to induce subtyping.
- **Not a type**: trait names can not be used as types.

  `This` can be used to refer to the eventual class of the code.
- **State**: No explicit constructors/fields; state is implicitly defined.
- **Composition**: traits code can be composed to obtain more code.

This leaking in our approach

```
IA={interface
   method Int ma()
   }
Utils={
   static method Int m(IA a){return ...}
   }
ta={implements IA
   method Int ma(){return Utils.m(this);}
   }
A=Use ta
B=Use ta
```

this written in trait `ta` is of type **This** and **IA**. when **B** inherits the code of `ta`, code `Uitls.m(this)` would pass a **this** as an **IA**. In this way, even if **B** and **A** share the same code, there are not in a subtype relation. (Note that `ta` is not a valid typename)

Trait composition

The composed code contains all members from the used traits. Methods with same name and type signature are summed into a single one. At most one of those summed methods can have a body, which will be propagated into the result.

```
t1={
  method String hello();
  method String helloWorld(){return hello()+" World";}
  }

t2={ method String hello(){return "Hi";} }

t3= Use t1,t2
//-- flatten to ---------------------------
t3= {
  method String hello(){return "Hi";}
  method String helloWorld(){return hello()+" World";}
  }
```

Handling State

How to make this process work with constructors and fields while achieving the following goals:

- managing fields in a way that borrows the elegance of summing methods;
- actually initializing objects, leaving no **`null`** fields;
- making it easy to add new fields;
- allowing self instantiation: a trait method can instantiate the class using it.

Abstract State Operations

Idea: use abstract methods as getters, setters and factory methods. Conventionally, an abstract class is a class with some abstract method. Here, a class whose abstract methods can be seen as state operations is a concrete coherent class.

For example, Java:

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

Equivalent code in 42_μ:

```
A={
  static method This of(Int x)
  method Int x()
  method Void x(Int that)
  }
```

Coherent class

- A class with no abstract methods is coherent (just like Java **`Math`**, for example). Such classes have no instances and are only useful for calling static methods.
- A class with a single abstract **`static`** method returning **`This`** and with parameters $T_1\ x_1, \ldots, T_n\ x_n$ is coherent if all the other abstract methods can be seen as *abstract state operations* over one of $x_1, \ldots, x_n$. That is:
  - A method $T_i\ x_i$**`()`** is interpreted as an abstract state method: a *getter* for x_i.
  - A method **`void`** x_i**`(`**T_i `that`**`)`** is a *setter* for x_i.

Example: **`Point`**s with algebraic operations

static factory method:

`Point p=Point.of(3,4)`

getters:

`p.x()==3`,   `p.y()==4`

summing points:

`p.sum(Point.of(10,20))` equivalent to `Point.of(13,24)`

NOTE: sum operation creates a new point

If no code reuse is desired, it is easy to use Java to encode such **`Point`** class. However, we would like to define operations `sum`, `mul`, `div` and `sub` independently and compose them to create classes with the operations we want, so that it is easy to have points with `sum` and `mul`, points with just `sum` or points with just `mul`.
For example, we expect `PointSum.sum(PointSum) : PointSum` but we also `PointSumMul.sum(PointSumMul) : PointSumMul`.
This breaks subtyping: **`PointSumMul`** $\not\leq$ **`PointSum`**.

Abstract state

```
p= { //point trait with abstract state
  method Int x() //getter for field x
  method Int y() //getter for field y
  static method This of(Int x,Int y) //factory method
  }

Point= Use p //this is a concrete coherent class
...
.. Point.of(3,7).x() ..//valid code
```

Abstract state

```
p={ method Int x()  method Int y()
  static method This of(Int x,Int y)
  }

pointSum= Use p, {
  method This sum(This that){
    return This.of(this.x()+that.x(),this.y()+that.y());
    }}

pointMul= Use p, {
  method This mul(This that){
    return This.of(this.x()*that.x(),this.y()*that.y());
    }}

pointDiv= ...

MyPoint= Use pointSum,pointMul,pointDiv,...
```

Case study 1: **`Point`**s with algebraic operations

Encoding the point example above, with the 4 arithmetic operations, and instantiating all the 16 possible permutations as separate classes:

Language	Lines of code	members	classes/traits
Java7	115	50	16
Classless Java	82	34	16
Scala	81	40	21
42_μ	32	7	21

State extensibility

```
p={ method Int x() method Int y()
  static method This of(Int x,Int y)
  }
pointSum= Use p, {
  method This sum(This that){
    return This.of(
      this.x()+that.x(),this.y()+that.y());
    }}

colored= { method Color color() }

CPoint= Use pointSum,colored,{
  static method This of(Int x,Int y,Color color)

  static method This of(Int x,Int y){
    return This.of(x,y,Color.of(/*red*/));
    } }//Now CPoint.sum return read points
```

State extensibility, with withers

A wither is like a field setter, but creates a new copy of the object

```
p= { method Int x()  method Int y()
  method This withX(Int that)
  method This withY(Int that)
  method This merge(This that)
  }
pointSum= Use p, {
  method This sum(This that){
    return this.merge(that)
      .withX(this.x()+that.x())
      .withY(this.y()+that.y());
  }}
colored= {  method Color color()
  method This withColor(Color that)
  method This merge(This that){
    return this.withColor(this.color().mix(that.color());
  }}
CPoint= Use pointSum,colored,{
  static method This of(Int x, Int y, Color color)}
```

Independent extensibility

```
flavored= {
  method Flavor flavor()
  method This withFlavor(Flavor that)
  method This merge(This that){
    return this.withFlavor(that.flavor());
  }}
FCPoint= Use //aliasing conflicting implementations
  colored[super merge as m1],
  flavored[super merge as m2],
  pointSum,{
    method This merge(This that){
      return this.m1(that).m2(that);
    }
    static method
    This of(Int x, Int y, Color color, Flavor flavor)
    }
```

More in the paper

- Transparent handling of nested classes
- `This` type generalized for family polymorphism
- A very natural encoding of the Expression Problem
- Simple formalization of our language

Thanks

Questions?