Separating Use and Reuse to Improve Both

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

[1] Victoria University of Wellington, New Zealand,
marco.servetto@ecs.vuw.ac.nz,
WWW home page: https://ecs.victoria.ac.nz/Main/MarcoServetto
[2] The University of Hong Kong, Hong Kong

Abstract. Abstract here! . . .

Keywords: Code Reuse, Object-Oriented Programming

1 Introduction

Historically, we have seen a lot of focus on the importance of separing subtype from subclassing [**?**]. This is claimed to be good for code reuse, design and reasoning. While there is no problem in subtyping without subclassing, in most OO languages code reuse (inheritance/extends) imply subtyping. Consider the following (Java) code:

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

This code seems correct, and there is no subtyping/subclassing. Now, lets add a class `B`

```
class B extends A{ int mb(){return this.ma();} }
```

Class `B` now have a method `ma` inherited from `A`. This method passes **`this`** as `A`; we can see an invocation of `ma` inside `mb`, where **`this`** is of type `B`. The execution will eventually call `Util.m` with a `B` instance. This can be correct only if `B` is a subtype of `A`. Thus, every OO language with the minimal features exposed in this example (using this, extends, method call) is forced to accept that subclassing implies subtyping. Note how this is true also in C++, where is possible to ”extends privately”. Such is a limitation of the visibility of subtyping but not over subtyping itself, and the former example would be accepted by C++ even if `B` was to ”privately extends” `A`. MARCO: Cite some work of bruceTHisType and show how he also fails. BRUNO: I was expecting some text/argument here on why subclassing implies subtyping is bad. Perhaps another example where subclassing must not imply subtyping to type-check.

We will show here a simple design to completley uncouple subtyping and subclassing in a nominally typed OO language, by dividing code designed for **USE** from code designed for **REUSE**.[3] First we show a minimal language, then we show how mutually recursive types are supported, how state/constructors/fields are supported, how we can extend the language with nested classes. Finally we show 42, a full blown language build around our ideas of reuse.

Our design leverage on traits []: a well know mechanisms for pure code reuse

[3] We need to talk of unanticipated extensions?

2 A Decoupled Trait Calculus

In order to illustrate how we plan to properly divide code reuse/inheritance/subclassing from subtyping, lets consider a simple language where trait names start lowercase $t$and class names start uppercase C. The syntax of the language is:

TD	$::= t\,\mathtt{:}\,L \mid t\,\mathtt{:}\mathtt{\textbf{Use}}\ \overline{V}$	Trait Decl
CD	$::= C\,\mathtt{:}\,L \mid C\,\mathtt{:}\mathtt{\textbf{Use}}\ \overline{V}$	Class Decl
V	$::= t \mid L$	Code Value
L	$::= \mathtt{\textbf{interface}}?\ \mathtt{\textbf{implements}}\overline{T}\ \overline{MD}$	Code Literal
T	$::= C$	types are class names
MD	$::= \mathtt{\textbf{class}}?\ \mathtt{\textbf{method}}\ T\ m\,\mathtt{(}\,\overline{T\,x}\,\mathtt{)}\ e?$	Method Decl
e	$::= x \mid T \mid e\,\mathtt{.}\,m\,\mathtt{(}\overline{e}\mathtt{)}$	expressions
D	$::= CD \mid TD$	Declaration

To declare a trait TD or a class CD, we can use either a code literal L or a trait expression. Traits come with various operators (restrict, hide, alias) but for now we focus on the single operator **`Use`**, taking a set of code values: that is trait names t or literals L and composing them in an *associative* and *commutative* way. This operation, sometimes called *sum*, is the simplest and most elegant composition operator. **`Use`** $\overline{V}$ composes the content of $\overline{V}$ by taking the union of the methods and the union of the implementations.

Use can not be applied if multiple versions of the same method are present in different traits. An exception is done for abstract methods: methods where the implementation e is missing; In this case (if the headers are compatible) the implemented version is selected. In a sum of two abstract methods with compatible headers, the one with the more specific type is selected. We will discuss later the formal details of Sum.MARCO: actually, this makes the sum not strongly associative...

Code literals L can be marked as interfaces. We will call class interface, or simply interface a class declaration of for `C:{`**`interface`**` ...}`. Then we have a set of implemented interfaces and a set of member declarations; in this simple language, just methods. If there are no implemented interfaces, in the concrete syntax we will omit the **`implements`** keyword.

Methods MDcan be instance methods or **`class`** methods. A class method is similar to a **`static`** method in Java but can be abstract. This is very usefull in the context of code composition. To denote a method as abstract, instead of an optional keyword we just omit the implementation e.

A version of this language where there are no traits can be seen as a restriction/variation of FJ [].

Basic well formedness rules applies:

- all the traits and classes have unique names in a program $\overline{D}$,
- all method parameters have unique names and the special parameter name **`this`** is not declared in the parameter list,
- all methods in a code literal have unique names,
- all used variables are in scope.

Those rules can be applied on any given Lindividually and in full isolation.

We expect the type system to enforce

– subtyping between interfaces and classes,
– method call typechecking,
– no circular implementation of interfaces,
– type signature of methods from interfaces can be refined following the well known variant-contravariant rules,
– only interfaces can be implemented.
– MARCO: I'm sure I'm missing something

We will see later the details on when exactly in the compilation process any code literal is typechecked. While classes are typed assuming **this** is of the nominal type of the class, trait declarations, do not introduce any nominal type. **this** in a trait is typed with a special type **This** that is visible only inside such trait. Syntactically, **This** is just a special, reserved, class name C. A Literal can use the **This** type, and when the trait is reused to create a class, **This** will become just an alias for the class name.

For the sake of simplicity, method bodies are just simple expressions e, whose can be just variables and method calls. We do not think that there is any problem to extend our language with other constructs; we use minimal expressions to keep our example simple and compact. In order to make our examples more engaging, we feel free to use the type int, numeric literals and their operations.

2.1 Code Use and Code Reuse in DTrait

The great absent in our language is the **extends** keyword. In a sense, all classes are final (as in Java). Moreover **Use** can not contain class names. That is, using a trait is the only way to induce code reuse.

Lets now try to encode the example in Section **??** e in our language:

```
A:{ method int ma() Utils.m(this) }//note, no {return _}
Utils:{ class method int m(A a)/*method body here*/ }
```

This code is fine, but there is no way to add a `B` reusing code of `A`, since `A` is designed for code *use* and not *reuse*. Lets try again, but this time aiming at code reuse:

```
ta:{ method int ma() Utils.m(this) }//type error
A:Use ta
Utils:{ class method int m(A a)/*method body here*/ }
```

This does not work, because `Utils.m` requires an `A` and **this** in `ta` have no knowlegde of `A`. Lets try again

```
IA:{interface method int ma()}//interface with abstract method
ta:{implements IA
  method int ma() Utils.m(this) }
A:Use ta
Utils:{ class method int m(IA a)/*method body here*/}
```

This code works: `Utils` is using an interface `IA` and the trait `ta` is implementing it. It is also possible to add a `B` as follow

```
  B:Use ta, { int mb(){return this.ma();} }
```

This also works. `B` reuses the code of `ta`, but has no knowledge of `A`. Since `B` reuses `ta`, and `ta` implements `IA`, also `B` implements `IA`.

Semantic of Use Abeit alternative semantic models for traits [] have been proposed, here we use the flattening model. This means that

```
A:Use ta
B:Use ta, { int mb(){return this.ma();} }
```

reduces/is equivalent to/is flatted into

```
A:{implements IA method int ma() Utils.m(this) }
B:{implements IA
  method int ma() Utils.m(this)
  int mb() this.ma() }
```

This code seems correct, and there is no mention of the trait `ta`. In some sense, all the information about code reuse/subclassing is just a private implementation detail of `A` and `B`; while subtyping is part of the class interface.

2.2 Remarks on Typing

Our typing discipline is what distinguish our approach from a simple minded code composition macro(cite) or a rigid module composition(cite may be some of my work).

The are two core ideas:

traits are well-typed *before being reused.*
For example in

```
t:{int m() 2
   int n() this.m()+1}
```

`t` is well typed since `m()` is declared inside of `t`, while

```
t1:{int n() this.m()+1}
```

would be ill typed.
Note how in the former example `ta` is well typed just by knowing itself and `IA`.

class expressions are not required to be well-typed *before flattening.*
In class expressions **`Use`** $\overline{V}$ an L in $\overline{V}$ is not well typed before flattening, only the result is supposed to be well typed. While this seams a very dangerous approach at first, consider that also Java have the same behaviour: for example in

```
  class A{ int m() {return 2;}  int n(){return this.m()+1;} }
  class B extends A{ int mb(){return this.ma();} }
```

in `B` we can call **`this`**`.ma()` even if in the curly braces there is no declaration for `ma()`.
In our example,using the trait `t` of before

```
C: Use t {int k() this.n()+this.m()}
```

would be correct: even if n,m are not defined inside `{int k() this.n()+this.m()}`, the result of the flattening is well typed.

This is not the case in many similar works in literature [] where the literals have to be self complete. In this case we would have been forced to declare abstract methods `n` and `m`.

Our typing strategy have two important properties:

– if a class is declared by using C : **`Use`** $\overline{t}$, that is, without literals, and the flattening is successfull, C is well typed, no need of further checking.
– on the other side, if a class is declared by C : **`Use`** $\overline{V}$, with $L_1 \ldots L_n \in \overline{V}$, and after successfull flattening $C : L$ is not well typed, the error was originally present in one of $L_1 \ldots L_n$. This also means that as an optimization strategy we may remember what method bodies come from traits and what method bodies come from code literals, in order to typecheck only the latter.

We will see now how this idea that code literals wrote in the declaration of a class do not have to be well typed on their own also lets us reason about mutually recursive types.

3 Recursive types

OO language leverage on recursive types most of the times. For example in a pure OO language, `String` may offers a `Int size()` method, and `Int` may offer a `String toString()` method.

This means that is not possible to type in (full) isolation classes `String` and `Int`.

The most expressive compilation process may divide the classes in groups of mutually dependent classes. Each group may also depend from a number of other groups. This would form a Direct Aciclyc Graph of groups. To type a group, we first need to type all depended groups, then we can extract the structure/signature/structural type of all the classes of the group. Now, with the information of the depended groups and the one extracted from the current group, it is possible to typecheck the implementation of each class in the group.

In this model, it is reasonable to assume that flattening happens group by group, before extracting the class signatures.

Here we go for a much simpler simple top down execution/interpretation for flattening, where flattening happen one at the time, and classes are typechecked where their type is first needed.

For example

```
A:{int ma(B b) b.mb()+1}
tb:{int mb() 2}
tc:{int mc(A a,B b) a.ma(b)}
B: Sum tb
C: Sum tc, {method int hello() 1}
```

In this scenario, since we go top down, we first need to generate B. To generate B, we need to use tb; In order modularly ensure well typedness, we require tb to be well typed at this stage. If tb was not well typed a compilation error could be generated at this stage. In this moment, A can not be compiled/checked alone, we need informations about B, but A is not used in tb, thus we do not need to type A and we can type tb with the available informations and proceed to generate B. Now, we need to generate C, and we need to ensure well typedness of tc. Now B is alreay well typed (since generated by Sum tb, with no Ls), and A can be typed; finally tc can be typed and used. If sum could not be performed (for example it tc had a method hello too) a compilation error could be generated at this stage.

On the opposite side, if B and C was swapped, as in C: Sum tc, method int hello() 1 B: Sum tb now the first task would be to generate C, but to type tc we need to know the type of A and B. But they are both unavailable: B is still not computed and A can not be compiled/checked alone, without information about B. A compilation error would be generated, on the line of ”flattening of C requires tc, tc requires A,B, B is still in need of flattening”.

In this example, a more expressive compilation process could compute a dependency graph and, if possible, reorganize the list, but for simplicity lets consider to always provide the declarations in the right order, if one exists.

Some may find the requirement of the existence of an order restrictive; An example of a ”morally correct” program where no right order exists is the following:

t: int mt(A a) a.ma() A:Sum t int ma() 1

We expect two (related) criticism to our compilation model: 1 In a system without inference for method types, if the result of composition operators depends only on the structural shape of their input (as for Sum) is indeed possible to optimistically compute the resulting structural shape of the classes and use this information to type involved examples like the former. We believe such typing discipline could be fragile, and could make human understanding the code reuse process much harder/involved. 2 In the world of strongly typed languages we could be tempted to first check that all can go well, and then perform the flattening. This would however be overcompicated for no observable difference: Indeed, in the former example there is no difference between First checking B and produce B code (that also contains B structural shape), then use B shape to check C and produce C code or a more involved First check B and discover B structural shape as result of the checking, then use B shape to check C. Finally produce both B and C code.

4 Managing State

The Sum operator is very elegant, but our research community is struggling to make it work with constructor/state/fields. The goals fo this struggle are as follow:

- keep sum associative and commutative.
- allowing a class to create instances of itself.
- actually initialize objects, not leaving no null fields
- managing fields in a way that borrows the elengance of summing methods
- make easy to add new fields

In the related work we will show some alternative ways to handle state. However the purest solution just requires methods: The idea is that the trait code just uses getter/setters/factories, while leaving to classes the role to finally define the fields/constructors. That is, the the class has syntax richer that the trait one, allowing declaration for fields and constructors. This approach is very powefull[see class less java]

Pro: This approach is associative and commutative, even self construction can be allowed if the trait requires a static/class method returning This; the class will then implement this method by forwarding a call to the constructor. Negative: writing the class code with the constructors and fields and getter/setters and factories can be quite tedious. There is no way for a trait to specify a default value for a field, the class need to handle all the state, even state that is conceptually ”private” of such trait.

for example

```
pointSum: { method int x(); method int y();
  static method This of(int x,int y);
  method This sum(This other)
    This.of(this.x+other.x,this.y+other.y);
  }
pointMul: { method int x(); method int y();
  static method This of(int x,int y);
  method This mul(This other)
    This.of(this.x*other.x,this.y*other.y);
  }
```

4.1 A First Attempt at Composition

```
Point:Use pointSum,pointMul
```

would fail since methods x,y and of are still abstract. In this model the user is required to write something similar to

```
CPoint:Use point,colored, {
  int x; int y;
  method int x()x; method int y()y;
  class method This of(int x, int y)
    new Point(x,y);
  constructor Point(int x, int y){
    this.x=x;this.y=y;this.color=color;
    }
  }
```

after a while programming in this style, those ”fixpoint” close the state in the obvious way classes becomes quite repetitive, and one wonder if it could be possible to automatically generate such code[] In our model we go one step further: In our model there is no need to generate code, or to explicitly write down constructors and fields; there is not even syntax for those constructor. The idea is that any class that ”could” be completed in the obvious way is a complete ”coherent” class. Others call abstract a class that have abstract methods. We call abstract a class that have a set of abstract methods that are not coherent, that is, you can not interpret those as factory,getters and setters.

In the detail: a class with no abstract method is coherent, and like Java Math will just be usefull for calling class/static methods. a class with a single abstract class method returning This is coherent if all the other abstract methods can be seen as ”abstract state operations” over one of its argument. For example, if there is a class method This of(int x, int y) as before, then a method int x() is an abstract state method, getter for x. a method Void x(int that) is abstract state method, setter for x. while getters and setters are fundamental operations, we can immagine more operations to be supported:

```
method This withX(int that)
```

may be a ”wither”, doing a functional field update method Void update(int x,int) may do two field update at a time method This clone() may do a shallow clone of the object. We are not sure what is the best set of abstract state operations yet.

lets play with the points of before, to see what good can we do with the current instruments:

```
pointSum: { method int x(); method int y();
  static method This of(int x,int y);
  method This sum(This other)
    This.of(this.x+other.x,this.y+other.y);
  }
pointMul: { method int x(); method int y();
  static method This of(int x,int y);
  method This mul(This other)
    This.of(this.x*other.x,this.y*other.y);
  }
PointAlgebra:Sum pointSum,pointMul
```

As you can see, we can declare the methods independently and compose the result as we wish. However we have to repeat the abstract methods x,y and of. In addition of Sum,Mul we may want many operations over points; can we improve our reuse and not repeat such abstract definitions? of course!

```
p: { method int x(); method int y();
  static method This of(int x,int y);
  }
pointSum:Sum p, { method This sum(This other)
    This.of(this.x+other.x,this.y+other.y);
  }
pointMul:Sum p, { method This mul(This other)
    This.of(this.x*other.x,this.y*other.y);
  }
pointDiv: ...
PointAlgebra:Sum pointSum,pointMul,pointDiv,...
```

now our code is fully modularized, and each trait handle exactly one method.

What happens if we want to add fields instead of just operations?

```
colored:{ method Color color() }
Point:Sum pointSum,colored
```

This first attempt does not work: the abstract color method is not a getter for any of the parameters of

```
class method This of(int x,int y)
```

A solution is to provide a richer factory:

```
CPoint:Sum pointSum,colored,{
  class method This of(int x,int y) This.of(x,y,Color.of(/*red*/)
  class method This of(int x, int y,Color color)
  }
```

where we assume to support overloading on different parameter number. This is a good solution, we think is better that any alternatives in literature, however the method CPoint.sum reset the color to red. What should be the behaviour in this case? If instead of writing This.of() we used this.withX(newX).withY(newY) we could preserve the color from this. Not sure if that would be better. If the point designer could predict this kind of extension, then we could use the following design:

```
p: { method int x(); method int y();
  method This withX(int that);
  method This withY(int that);
  static method This of(int x,int y);
  method This merge(This other);
  }
pointSum:Sum p, { method This sum(This other)
    this.merge(other).withX(this.x+other.x).withY(this.y+other.y);
  }
colored:{method Color color();
  method This withColor(Color that);
  method This merge(This other)
    this.withColor(this.color().mix(that.color());
  }
CPoint:/*as before*/
```

Now we can merge colors, or any other kind of state we may want to add following this pattern.

5 Related Work

6 Conclusions