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

In mainstream OO languages like Java, C++ or C# subclassing implies subtyping. For example, in Java a subclass definition, such as:

```
class A extends B {}
```

does two things at the same time: 1) **inheriting** code from class `B`; and **creating a subtype** of *B*. Therefore in a language like Java a subclass is *always* a subtype of the extended class.

Historically, there has been a lot of focus on the importance of separing subtyping from subclassing [**?**]. This is claimed to be good for code reuse, design and reasoning. There are at least two distinct situations where the separation of subtyping and subclassing is helpful.

- **Allowing inheritance/reuse even when subtyping is impossible:** In some situations a subclass contains methods whose signatures are incompatible with the superclass, yet inheritance is still possible. A typical example, which was illustrated by Cook et al., are classes with *binary methods* [].
- **Preventing unintended subtyping:** For certain classes we would like to inherit code without creating a subtype even if, from the typing point of view, subtyping is still possible. A typical example [] of this are methods for collection classes such as Sets and Bags. Bag implementations can often inherit from Set implementations, and the interfaces of the two collection types are similar and type compatible. However, from the logical point-of-view a Bag is *not a subtype* of a Set.

Type systems based on structural typing can deal with the first situation well, but not the second. Since structural subtyping accounts for the types of the methods only, a Bag would be a subtype of a Set if the two interfaces are type compatible. For dealing with the second situation nominal subtyping is preferable. With nominal subtyping an explicit subtyping relation must be signalled by the programmer. Thus if subtyping is not desired, the idea is that programmer can simply not declare a subtyping relationship.

While there is no problem in subtyping without subclassing, in the design of most nominal OO languages subtyping implies subtyping in a fundamental way. This is because of what we call the self-reference *leaking problem*. The leaking problem is illustrated by the following (Java) code:

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

In class `A`, the method `m` passes **`this`** as `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();} }
```

We can see an invocation of `ma` inside `mb`, where the self-reference **`this`** is of type `B`. The execution will eventually call `Util.m` with an instance of `B`. However, this can be correct only if `B` is a subtype of `A`.

If Java was to support a mechanism to allow reuse/inheritance without introducing subtyping, such as:

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

Then an invocation of `mb` would be type-unsafe (i.e. it would result in a run-time type error). Note that here the intention of using the imaginary keyword **inherits** is to allow the code from `A` to be inherited without `B` becoming a subtype of `A`. However this breaks type-safety. The problem is that the self-reference **`this`** in class `B` has type `B`. Thus, when **`this`** is passed as an argument to the method `Utils.m` as a result of the invocation of `mb`, it will have a type that is incompatible with the expected argument of type `A`. BRUNO: We can also mention what happens in structural types: "System with structural types would *be required to* guarantee that the structural type of A is a supertype of the structural type of B.

What the leaking problem illustrates is that adopting a more flexible nominally typed OO model where subclassing does not imply subtyping is not trivial. If we are to separate subclassing from subtyping then a more substantial change in the language design is necessary. Every OO language with the minimal features exposed in the example (using **`this`**, extends and method calls) is forced to accept that subclassing implies subtyping.[3] In essence we believe that a problem with the design of classes in languages like Java is that classes do too many things at once. In particular they act both as units of *use* and *reuse*. That is, in languages like Java, classes can be instantiated (using **new**) to create objects that *use* some functionality. Classes can also be subclassed to provide *reuse* of code.

MARCO: Cite some work of bruceTHisType and show how he also fails.

This paper aims at showing a simple language design, called **DTrait**, to completely decouple subtyping and subclassing in a nominally typed OO language. The key idea is to divide between code designed for **USE** and code designed for **REUSE**. In **DTrait** there are two separate concepts: classes and traits []. Classes are meant for code use, and cannot be used for reuse. In some sense classes in **DTrait** are like final classes

[3] 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`.

in Java. Traits are meant for code reuse and multiple traits can be composed to form a class which can then be instantiated. Traits cannot be instantiated (or used) directly. Such design allows the subtyping and code reuse to be treated separately, which in terms brings several benefits in terms of flexibility and code reuse. MARCO: We need to talk of unanticipated extensions? BRUNO: Talk more about typing aspects here. Summarize the important aspects of the design of **DTrait**.

First we show a minimal language that illustrates the basic ideas. 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.

1.1 List of contribution

MARCO: text to improve, but is nice to keep a list

- simple approach to fully separate inheritance and subtyping
- easy to grasp model of compilation were typing is late but not too late (lazy?)BRUNO: this idea is important: it's a key design decision. We need to talk about it in the introduction and offer some motivation/justification.
- clean and elegant handling of state in trat/module composition language
- extension with nested classes is natural and powefullBRUNO: wondering if this is necessary. I think there are several interesting/novel ideas already. Covering too much will result on not saying enough about each individual ideas.
- more operators can be accomodated without changing the general model

2 The Design of DTrait: Separating Use and Reuse

This section presents the overview of **DTrait**, and it illustrates the key ideas of its design. In particular we illustrate how to separate code use and code reuse in **DTrait**.

2.1 Classes in DTrait: A mechanism for code Use

The concept of a class in **DTrait** provides a mechanism for code use only. This means that in **DTrait** there is actually no subclassing, and classes are roughly equivalent to final classes in Java. Thus, compared to Java-like languages, the most noticeable difference is the absence of the **`extends`** keyword in **DTrait**.

To illustrate classes in **DTrait** consider the example in Section **??**:

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

Classes in **DTrait** use a slightly different declaration style compared to Java. In **DTrait** there is no **`class`** keyword. Class declarations have a name (which must always start with an uppercase letter) and a code literal, which is used to specify the definitions of the class. For instance, in the class declaration for `A`, the name of the class is `A` and the code

literal associatted with the class (`{ method int ma() Utils.m(this) }`) contains the definitions associated to the class. There are, however, some important differences to Java-like languages in the way classes and code-literals are type-checked, as we shall see next. Nevertheless, for this example, things still work in a similar way to Java.

The **DTrait** code above is fine, but there is no way to add a class `B` reusing the code of `A`, since `A` is designed for code *use* and not *reuse*. So, unlike the Java code, introducing a subclass `B` is not possible. At first, this may seem like a severe restriction, but **DTrait** has a different mechanism for code *reuse* that is more appropriate when code reuse is intended.

2.2 Traits in DTrait: A mechanim of code Reuse

Unlike classes traits in **DTrait** cannot be instantiated and do not introduce new types. However they provide code reuse. Trait declarations look very much like class declarations, but trait names start with a lowercase letter. An obvious first attempt to model the example in Section **??** (for a Java programmer) with traits and code reuse is:

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

Here `ta` is a trait intended to replace the original class `A` so that the code of the method `ma` can be reused. Then the class `A` is created by inheriting the code from the trait using the keyword **Use**. Note that **`Use`** cannot contain class names: only trait names are allowed. That is, using a trait is the only way to induce code reuse. Unfortunatelly, this code does not work, because `Utils.m` requires an `A` and the type of **`this`** in `ta` has no relationship to the type `A`. Since the trait name is not a type, no code external to that trait can refer to it. This is one of the key design decisions in **DTrait**. The point is that the trait can refer to the program via **`this`**, but the program is agnostic to what the trait type is going to be, so it can be later assigned to any (or many) classes. This improves the flexibility of reuse as illustrated in Section **??**. However, to solve the more immediate typing issue above we need one more round of refactoring, as we shall see next.

Type of the self-reference The code above does not work because the way type-checking works in **DTrait** is that type-checking of code literals is independent of the class/trait names associated to it. A Java programmer may expect that the type of **`this`** in the previous definition of `ta` is `ta` itself. However, this intuition brought from Java is wrong in **DTrait** for two reasons:

- The first reason is that **traits are not types**. Traits in **DTrait** are simply units of reuse and cannot be used as types. Types are only introduced by class/interface declarations.
- The second reason is that **the type of self-references is the self type of the code literal.** Unlike Java-like languages, where the body of a class declaration is intrinsically coupled with the class itself, in **DTrait** code literals are first-classBRUNO: Is it appropriate to say that they are first class? and are type-checked independently. **DTrait** has a notion of self-type, which is closely related to approaches such as

	Instantiable (Use)	Unit of Reuse	Introduces New Type
Class	Yes	No	Yes
Trait	No	Yes	No

Fig. 1. A comparision between traits and classes in **DTrait**.

ThisType [] and ...[**?**]. Therefore in **DTrait** the self-reference does not have the type of the class being defined, but rather it has the self-type.

With this in mind, we can try to model the example in Section **??** again:

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

This code works: `Utils` is using an interface `IA` and the trait `ta` is implementing it. Thus the self-type of the trait `ta` (and the type of **`this`**) will be a subtype of `IA`. It is also possible to add a `B` as follows

```
B:Use ta, { method 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 Albeit alternative semantic models for traits [] have been proposed, here we use the flattening model. This means that

```
A:Use ta
B:Use ta, { method 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)
  method 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.

To finish this section, Figure 1 provides a summary of the differences between classes and traits. The comparison focus on the roles of traits and classes with respect to instantiation, reusability and whether the declarations also introduce new types or not.

3 Improving Use by Preventing Unintended Subtyping

As stated in Section **??** a possible benefit of a nominally typed OO language that separates inheritance from subtyping is that it is possible to have reuse, while preventing

unintended subtyping. The design of **DTrait** allows this, thus it *improves the use* of classes, while retaining the benefits of reuse when compared to Java-like languages. Here we illustrate this benefit by modelling a simplified version of Set and Bag collections.

BRUNO: Plan: Show simplified Java code. Show how the Java code forces us to make Bags a subtype of Set, if we want reuse. Then show the 42/language in this paper/ code and solution.

An iconic example on why connecting inheritance/code reuse and subtpying is problematic is provided by the historic[**?**]: A reasonable implementation for a `Set` may be easy to extend into a `Bag` keeping tracks of how many times an element occurs. We would just add some state and override a couple of methods.BRUNO: Are we going to present this example in the paper solved in 42, for example? I think I would expect to see it.

However, our subclassing would break Liskov substitution principle (LSP) [**?**]: not all bags are sets! Of course, one could retroactivelly fix this problem by introducing `AbstractSetOrBag` and making both `Bag` and `Set` inherit from it. This looks unnatural, since `Set` would extend it without adding anything, and we would be surprised to find a use of the type `AbstractSetOrBag`. Worst, if we was to constantly apply this mentalty, we would introduce a very high number of abstract classes that are not supposed to be used as types, and that will clutter the public interface of our classes and our code project as a whole.

```
class Set {.../* Elem put() boolean isIn(Elem)*/ ...}
class Bag extends Set{.../*override something to keep track of duplications*/..}
```

but now,

```
Set mySet=new Bag(); //OK for the type system but not for LSP
```

as a pleasaruble accident, avoid such code gift us simple support for This type and (in the extensions with nested classes seen later) family polimporphism.

4 Improving Reuse when Subtyping is Impossible

DTrait retains the ability to allow reuse even when subtyping is impossible. That is, in some situations a subclass contains methods whose signatures are incompatible with the superclass, yet inheritance is still possible. This type of situations is the primary motivator for previous work aiming at separating inheritance from subtyping, and it happens, for example, in classes with *binary methods* []. Despite the adoption of a nominal approach and unlike other nominally typed languages, the leaking problem does not prevent **DTrait** from enjoying such kind of improved reuse.

To illustrate how **DTrait** allows for improved reuse, we consider a pattern to combine *multiple trait inheritance* and *state*.

4.1 Managing State

This idea of summing pieces of code is very elegant, and has proven very successfull in module composition languages [**?**] but our research community is struggling to make it work with object state (constructor and fields) while acheving the following goals:

- keep sum associative and commutative,
- allowing a class to create instances of itself,
- actually initialize objects, 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 [**?**]

Advantages: 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.

Negatives: writing the class code with the constructors and fields and getter/setters and factories can be quite tedious. Moreover, 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.

Here in the following an exaple of such approach:

```
pointSum: { method int x()  method int y()
  class 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()
  class method This of(int x,int y)
  method This mul(This other)
    This.of(this.x*other.x,this.y*other.y)
  }
```

As you see, all the state operations are represented as abstract methods.

4.2 A First Attempt at Composition

According to the general ideas expressed before,

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

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

```
CPoint:Use pointSum,pointMul, {//not our suggested solution
  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 }
  }
```

after a while programming in this style, writing those obvious "close the state" classes become a repetivie boring job, and one wonder if it could be possible to automatically generate such code [**?**]. Indeed those classes are just a form of "fixpoint".

In our model we go one step further: 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*. In most other languages, a class is abstract if have abstract methods. Instead, we call abstract a class whose set of abstract methods is not coherent, that is, can not be automatically recognized as factory,getters and setters.

Detaild definition of coherent:

- 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 intepreted as an abstract state method: a getter for `x`.
- a method `Void x(`**`int`** `that)` is a setter for x.

While getters and setters are fundamental operations, we can immagine more operations to be supported; for example

- **`method This`** `withX(`**`int`** `that)` may be a "wither", doing a functional field update.
- **`method`** `Void update(`**`int`** `x,`**`int`** `y)` 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, but we think this could become a very interesting area of research.

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

```
//same code as before for pointSum and pointMul
 pointSum: { method int x() method int y()
   class 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()//look we are repeating
   class method This of(int x,int y)//the abstract method declarations.
   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` and `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()
  class method This of(int x,int y)
  }
pointSum:Use p, { method This sum(This other)
    This.of(this.x+other.x,this.y+other.y)
  }
pointMul:Use p, { method This mul(This other)
    This.of(this.x*other.x,this.y*other.y)
  }
pointDiv: ...
PointAlgebra:Use 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 //fails
```

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:Use 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` resets the color to red. What should be the behaviour in this case? If we support withers, instead of writing `This.of()` we can use `this.withX(newX).withY(newY)` in order to preserve the color from `this`. Sadly, if we use this design inside of `sum(This other)` we would loose the color from `other`.

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:Use 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. In order to compose, let say `colored` with `flavored` we would need to compose the merge operation inside of both of them. The simple model we are presenting could accomodate this with an extension allowing code literals inside of a **`Use`** expression to use some form of super call to compose conflicting implementations. This is similar to the *override* operation present in the original trait model [**?**].

5 Old: Self Types, Binary Methods and the Need to Separate Inheritance from Subtyping

So, far it appears that we have not gained much from decoupling reuse from subtyping.

As discussed in Section **??** the type of the self-reference is the self-type. Like languages with ThisType, **DTrait** allows explicit references to the self type. This is useful in several situations: For example it can be helpful in the presence of binary methods [], or methods that return an instance of the class being defined.

6 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 with a lowercase $t$and class names start with an uppercase C. The syntax of the language is:

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. In our simple modeBRUNO: you mean model?, we consider an error trying to merge a class and an interface.

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 **`Use`**.MARCO: actually, this makes the sum not strongly associative... BRUNO: ok: would be nice to see an example?

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, the only members are methods. If there are no implemented interfaces, in the concrete syntax we will omit the **`implements`** keyword.

Methods MD can 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 [**?**].

Well-formedness Basic well formedness rules apply:

- 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 L individually 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.
- BRUNO: method conflict detection in traits?
- 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: they can be just variables and method calls. We do not think that there is any problem to extend our language with other constructs BRUNO: don't you actually have evidence that this is the case from 42?. 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.

6.1 Remarks on Typing

Our typing discipline is what distinguishes our approach from a simple minded code composition macro [**?**] or a rigid module composition [**?**]

The are two core ideas:

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

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

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

```
t1:{method 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`.BRUNO: why IA? it is not used in this example, right?

class expressions are not required to be well-typed *before flattening.*
In class expressions **Use** $\overline{V}$ an L in $\overline{V}$ is not typechecked before flattening, and only the result is expected to be well-typed. While this seems 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 {method int k() this.n()+this.m()}
```

would be correct: even if n,m are not defined inside `{method 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`.BRUNO: right. this is a good point

Our typing strategy has 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 $L_1 \dots L_n \in \overline{V}$, and after successfull flattening $C : L$ can not be typechecked, then the issue was originally present in one of $L_1 \dots L_n$. It may be that the result is intrinsically ill-typed, if of the methods in $L_1 \dots L_n$ is not well typed, but it may also happen that a type referred from one of those methods is declared *after* the current class. As we will see later, this is how our relaxation allows to support recursive types.
  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.

7 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:{method int ma(B b) b.mb()+1}
tb:{method int mb() 2}
tc:{method int mc(A a,B b) a.ma(b)}
B: Use tb
C: Use 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 **Use**`tb`, with no L), and `A` can be typed; finally `tc` can be typed and used. If **Use** 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: Use tc, {method int hello() 1}
B: Use 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`, but `B` is still in need of flattening".

In this example, a more expressive compilation/precompilation 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.

Criticism: existence of an order is restrictive.
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:Use t {int ma() 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 **`Use`**) 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 stick to our simple approach, since we believe such typing discipline would be fragile, and could make human understanding the code reuse process much harder/involved. Indeed we just wrote a program where the correctness of trait `t` depends of `A`, that is in turn generated using trait `t`.

Criticism: it would be better to typecheck before flatteningi.
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 `A,B,C` example above there is no difference between

- (1)First check `B` and produce `B` code (that also contains `B` structural shape), (2) then use `B` shape to check `C` and produce `C` code; or a more involved
- (1)First check `B` and discover just `B` structural shape as result of the checking, (2)then use `B` shape to check `C`. (3) Finally produce both `B` and `C` code.

Note that we can reuse code only by naming traits; but our only point of relaxation is the class literal: there is no way an error can "move around" and be duplicated during the compilation process. In particular, our approach allows for safe libraries of traits and classes to be fully typechecked, deployed and reused by multiple clients: no type error will emerge from library code. On the other side, we do not enforce the programmer to write always self-contained code where all the abstract method definition are explicitly declared.

8 Managing State

This idea of summing pieces of code is very elegant, and has proven very successfull in module composition languages [**?**] but our research community is struggling to make it work with object state (constructor and fields) while acheving the following goals:

- keep sum associative and commutative,
- allowing a class to create instances of itself,
- actually initialize objects, 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 [**?**]

Advantages: 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.

Negatives: writing the class code with the constructors and fields and getter/setters and factories can be quite tedious. Moreover, 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.

Here in the following an exaple of such approach:

```
pointSum: { method int x()  method int y()
  class 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()
  class method This of(int x,int y)
  method This mul(This other)
    This.of(this.x*other.x,this.y*other.y)
  }
```

As you see, all the state operations are represented as abstract methods.

8.1 A First Attempt at Composition

According to the general ideas expressed before,

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

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

```
CPoint:Use pointSum,pointMul, {//not our suggested solution
  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 }
  }
```

after a while programming in this style, writing those obvious “close the state” classes become a repetivie boring job, and one wonder if it could be possible to automatically generate such code [**?**]. Indeed those classes are just a form of “fixpoint”.

In our model we go one step further: 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*. In most other languages, a class is abstract if have abstract methods. Instead, we call abstract a class whose set of abstract methods is not coherent, that is, can not be automatically recognized as factory,getters and setters.

Detaild definition of coherent:

- 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 intepreted as an abstract state method: a getter for `x`.
- a method `Void x(`**int** `that)` is a setter for x.

While getters and setters are fundamental operations, we can immagine more operations to be supported; for example

- **method This** `withX(`**int** `that)` may be a "wither", doing a functional field update.
- **method** `Void update(`**int** `x,`**int** `y)` 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, but we think this could become a very interesting area of research.

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

```
//same code as before for pointSum and pointMul
 pointSum: { method int x() method int y()
   class 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()//look we are repeating
   class method This of(int x,int y)//the abstract method declarations.
   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` and `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()
   class method This of(int x,int y)
   }
 pointSum:Use p, { method This sum(This other)
     This.of(this.x+other.x,this.y+other.y)
   }
 pointMul:Use p, { method This mul(This other)
     This.of(this.x*other.x,this.y*other.y)
   }
 pointDiv: ...
 PointAlgebra:Use 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 //fails
```

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:Use 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` resets the color to red. What should be the behaviour in this case? If we support withers, instead of writing `This.of()` we can use `this.withX(newX).withY(newY)` in order to preserve the color from `this`. Sadly, if we use this design inside of `sum(This other)` we would loose the color from `other`.

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:Use 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. In order to compose, let say `colored` with `flavored` we would need to compose the merge operation inside of both of them. The simple model we are presenting could accomodate this with an extension allowing code literals inside of a `Use` expression to use some form of super call to compose conflicting implementations. This is similar to the *override* operation present in the original trait model [**?**].

9 Extensions to our model

One of the main feature of our simple reuse/use model is that it can be easly extended. One simple but amazingly expressive extension is nested classes

9.1 Nested classes

A nested class will be another kind of member in the Literal, so the grammar can be updated as following:

$MD ::= \textbf{class}?\ \textbf{method}\ T\ m\ (\overline{T\ x})\ e? \mid CD$ Member Decl
$T \quad ::= C \mid C\,.\,T$ types are now paths

The general idea is that by composing code with **Use**, nested classes with the same name are recursivelly composed. Note that while we have nested classes, we do not have nested traits: all traits are still at top level. Untypable/unresolved Traits are also the only"dependency" the type system keeps track of, this means that when a nested class at an arbirary nested level is flattend, as in `C:{ D:{ E:Use t1,t2,L}}` t1 and t2 must be defined before C at top level; and they may require classes (and their nested) defined before C. This means that the type system can still consider the class table as a simple map from Types T to their definition.

This extension lets us challenge the expression problem [**?**]: in the expression problem we have data variants and operations and....

Let see how to easily encode and solve the expression problem:

```
exp:{Exp:{interface}}//Exp declared once, reused everywhere
lit:Use exp,{ Lit:{implements Exp //Exp not explicitly declared
    class method Lit of(int inner) //Lit abstract state
    method int inner()}
  }
sum:Use  exp,{ Sum:{implements Exp
    class method This of(Exp left, Exp right)
    method Exp left() //Sum abstract state
    method Exp right()}
  }
uminus:Use exp,{ UMinus:{implements Exp
    class method This of(exp inner)//and so on for
    method Exp inner()}//all the needed datavariants
  }

expToS:{Exp:{interface method String toString()}}
//concept of toString declared once

sumToS:Use sum,expToS,{ Sum:{implements Exp//with Exp.toString
    method String toString()//just the implementation of the
      left.toString()+"+"+right.toString()//specific method
  }
uminusToS:...//implement toString for all the datavariants

expEval:{Exp:{interface method int eval()}}
//declare the next operation and implement it for all the datavariant
```

Now that you have nicely modularized the code, just compose all the traits you need.

```
MySolution:Use sumToS,litToS
//sum,lit and exp traits are alread included
```

The expression problem presented up to now is the traditional challenge proposed by [**?**]; this has been criticized to not really address the fundamental issues since it does not handle .... Now we show how we can go behond the traditional expression problem by encoding transformer methods: For example, lets add 1 to all literals

```
expAdd1:{Exp:{interface method Exp add1()}}
sumAdd1:Use sum,expAdd1,{Sum:{implements Exp
    method Exp add1()
      Sum.of(left.add1,right.add1())
  }
litAdd1:Use lit,expAdd1,{Lit:{implements Exp
    method Exp add1()
      Lit.of(inner()+1);
    }

MySolutionAdd1:Use sumToS,litToS,sumAdd1,litAdd1
```

This nicely solve our problem. However, notice how if we wished to add many similar operations we would have to repeat the propagation code (as in `sumAdd1`) many times just changing the name of the operation. In the next section we will show how to improve on this point.

9.2 More composition operators

`Use` is amazing, elegant and simple, but our system can be easily enriched with more operators: while most approaches in literature presents a fixed set of operators, we do not need such restriction. We just need to be sure that every newly added operator respects the following criteria:

- As for **`Use`**, the operato does not need to be total, but if it fails it needs to provide an error that will be reported to the programmer.
- When the operator takes in input only traits (they are going to be well typed), if a result is produced, such result is well typed.
- When the operator takes in input also code literal, if a non well typed result is produced, the type error must be tracked back to code in one of those non typed yet code literals.

Here we see some example of other composition operators, inspired from [**?**]:

Restrict
Restrict makes a method abstract

```
t:{method bool geq(This x) x.leq(this)   method bool leq(This x) x.qeq(this) }
C:Use t[restrict geq],{method bool geq(This x)  /*actual geq impl*/}
```

A variant of this operator allows to move the implementation to another name. This is very usefull to implement **`super`**. This also is enough to suppor the example of before, when we try to compose `colored` with `flavoured`.

```
colored:{/*as before*/}
flavoured:{
method Flavour flavour()
  method This withFlavour(Flavour that)
  method This merge(This other)
    this.withFlavour(this.flavour().mix(that.flavour())
  }
CPoint:Use colored[restrict merge as _1merge],flavoured[restrict merge as _2merge],
pointSum,{
  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)
  method This merge(This other)
    this._1merge(other)._2merge(other)
  }
```

Note how we are leveraging on the fact that the code literal does not need to be complete, thus we can just call `_1merge()` and `_2merge()`

Rename
Rename allows to make some form of "compile time" refactoring There are a lots of different forms of rename in literature, sometime allowing only to rename specific methods, sometime allowing to rename nested classes into other nested classes either at the same or at a different nesting level. Renaming in the context of nested classes also means that when renaming a method of an interface, all the nested classes implementing such interface inside of that code literal need to be adjusted. Renaming need to rename not only the method headers, but all the method calls inside of method bodies. At first glimps, this seams to be not always possible since we are considering to be able to apply those operators also to non well typed code. However, if the expression language is simple enough, it is possible to pre process the code to annotate the expected receiver type on all method calls by doing a purelly sintactic analysis on a single code literal in isolation. All the expression whose type is guessed to be out of the border of the literal can stay unannotated; they are not going to be renamed anyway.

```
t:{ I:{interface method int mI() }
     A:{implements I  method int mI() 42}
     B:{ method int mB(I i, A a, C c) i.mI()+a.mI()+c.mI()}
     //mB would be annotated i[I].mI()+a[A].mI()+c.mI()
     }
D:t[rename A.mI kI]
```

Notice how we are sure that c does not implements I since it is invisible from the outside: traits does not introduce nominal types!

We expect the flattened version for `D` to be

```
D:{ I:{interface method int kI() }
     A:{implements I  method int kI() 42}
     B:{ method int mB(I i, A a, C c) i.kI()+a.kI()+c.mI()}
     }
```

Hide can be seen as a variation of rename, where the method/class is renamed to a fresh unguessable name.

Redirect
Redirect allows to emulate generics; the main idea is that a (fully abstract) class can be redirect to another one external to the trait/code literal. For example a linked list can be implement as

```
list:{ Elem:{}
     Cell:{class method Cell of(Elem e,Cell c)
       method Elem e()  method Cell c()
       }
   method Elem get(int x) ...
   ...more methods..
   }
ListString:list[redirect Elem to String]
```

Application on the expression problem
With redirect, rename and restrict we can have the general operator propagator

```
operation:{//for sum and lit, easy to extends as before
  T:{}
  Exp:{interface method Exp op(T x)}}
  Sum:Use sum,{ extends Exp sum,expAdd1,{
    method Exp op(T x)
      Sum.of(left.op(x),right.op(x))  }
  Lit:Use lit,{
    method Exp op(T x)  this
  }
```

Now, to have my addN I can

```
opAddn: Use
  operation[redirect T to Int]
    [rename Exp.op(x) to addN(x)][restrict Lit.op(x)], {
  Lit:{method Exp addN(Int x) Lit.of(inner())+x}
  }
```

Full power of redirect
An expressive form of Redirect can be multiple, that is, can redirect may interdependent classes at the same time. We show an example where a specific kind of `Service` can produce a `Report`, and `Report`s can be combined together. The goal is to execute a list of such services and produce a collated report. This example also show how to propagate generics:

```
Service:{interface method Void performService()}
serviceCombinator:{
  S:{implements Service method R report()  }

  R:{method R combine(R that)   class method R empty() }

  ListS:list[redirect Elem to S]
```

```
  class method R doAll(ListS ss){//here we use extended java like syntax
    R r=R.empty()
    for(S s in ss){
      s.performService();
      r=r.combine(s.report())
      }
    return r;
  }
}
PaintingService:serviceCombinator[redirect S to PaintingService]
PaintingService:{... method PaintingReport report()..}
PaintingReport:{..}
```

The flattened version of `PaintingService` would look like:

```
PaintingService:{
  ListS:/*the expansion of list[redirect Elem to PaintingService]*/

  class method PaintingReport doAll(ListS ss){
    PaintingReport r=PaintingReport.empty()
    for(PaintingService s in ss){
      s.performService();
      r=r.combine(s.report())
      }
    return r;
  }
}
```

Where you can note how redirect figured out `R=PaintingReport` by comparing the structural shape of classes `PaintingService` and `S`.

To encode the former generic code in java you need to write the following headeche inducing interfaces for RService and Report. and require that the services you want to serve implement those.

```
interface Service{ void performService();}
interface Report<R extends Report<R>>{R combine(R that);}
interface RService<R extends Report<R>> extends Service{ R report();}
```

Note how we still can not encode the method `empty`.

10 Related Work

The literature on code reuse is too vast to let us do justice of it in a fiew pages, Our work is clearly inspired on Traits [**?**]; they are in turn insipired on module composition languages [**?**] and multiple inheritance [].

Our idea of fully separate inheritance and subtyping has some precedence In all the line of work of Ferruccio [cite many] neither traits nor classes introduce nominal types, that are provided only by interfaces. In DeepFjig[] classes introduce nominal types but in an innatural way, the type of **`this`** is only **`This`** (sometimes called <>) and not the nominal type of its class.

Other work cite [**?**,**?**] do no completly break the relation between inheritance and subtyping, but only prevent subtyping where it would be unsound.

Our compilation model is a clear step forward with respect to [**?**,**?**, metaFjig, deepFjig,myThesis] where all code was required to declare all the abstract dependencies all of the time.

The idea of abstract state operation has emerged from CJ [**?**], and offer a clean solution to handling state in a trait composition setting. Note how abstract state operation are different from just hiding fields under getter and setters: in our model the programmer simply never declare what is the state of the class, not even what information is stored in fields. The state is computed by the system as an overall result of the whole code composition process.

In literature there has been many attempts to add state in traits/module composition languages.

- No constructor: all the fields start at null/a default specified value. In this model fields are like another kind of member, and two fields with identical type can be merged by sum; `new C()` can be used for all classes, and `init` method may be called later, as in `Point p=new Point(); p.init(10,30).`
  To its credit, this simple approach is commutative and associative and do not disrupt elegance of summing methods. However, objects are created "broken" and the user is trusted with fixing them. While is easy to add fields, the load of inizializing them is on the user; moreover all the objects are intrinsically mutable, so this model is unfriendly to a functional programming style.
- Constructor compose fields: In this approach the fields are declared but not initialized, and a canonical constrctor constructor taking a value for each field and just initializing such field is automatically generated in the resulting class.
  In this approach it is easy to add fields, however this approach is associative but not commutative, since the order of composition will change the order of the fields; moreover there is no support for self construction since the signature of the constuctor change during composition.
- Constructor can be composed if they offers the same exact parameters: In this approach traits declare field and constructors. The constructor initialize the fields but can do any other computation. Traits whose constructors have the same signature can be composed. The composed constructor will execute both constructor bodies in order.
  This approach is designed to allows Self construction. It is also associative and mostly commutative: the order of composition is relevant only in the order of execution of side effects during construction. However the requirement that the constructors need the same parameters hamper reuse, and if a field is added, its initial value need to be somehow syntetized from the old constructor parameters.

We claim that our solution to the expression problem is the most natural in literature to date: While a similar sintax can be achieved with the scandinavian style [**?**], their dependent type system makes reasoning quite complex, and indeed more recent solutions have accepted a more involved syntax in order to have an understandable type system [**?**].

Our close contendant is DeepFJig [**?**]: all our gain over their model is based on our relaxation over abstract signatures. Note how we solve the expression problem in the most radical way possible MARCO: bruno, here you can insert some of the criticism over expression problem trivially and explain how is not applicable to us here.

11 Conclusions and practical applications

In this paper we explained a simple model to radically decouple inheritance/code reuse and subtyping.

We show how our model can be easly extended with more operators, if they behave well with respect to a simple predicate.

In order to focus on the explanation of the high level concepts, this paper do not present formalism for our approach. Formalism for similar approaches can be found in[deefjig, myThesis].

We are working on 42: a full blown language that leverage on the ideas presented in this paper to obtain reliable and understandble metaprogramming. Formalization (in progress) for such language can be found at URLURLURL. 42 extands our model allowing flattening to execute arbitrary computations. In such model we do not need an explicit notion of traits: they are encoded as method returning a code literal.

In addition to this, 42 have feature less related to code composition, like a strong type system supporting aliasing mutability[] and circularity control[], checked exceptions[], errors (unchecked exceptions) with strong exception safety for errors[].

42 do not have a finite set of composion operators; they can be added using the built in support for native method calls. They can be dynamically checked to verify that they are well behaved according to our predicate, or they can be trusted to acheive efficiency.

Bibliography