Separating Use and Reuse to Improve Both

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

[1] Victoria University of Wellington, New Zealand,
[2] The University of Hong Kong, Hong Kong

Abstract. In most OO languages subclassing/inheritance implies subtyping. This is considered by many a conceptual design error, but it seems required for technical reasons, due to what we call the *this-leaking problem*. It shows that separating inheritance from subtyping is non-trivial and requires a significant departure from the OO models in existing mainstream OO languages. We are aware of at least 3 independently designed research languages addressing this limitation:Delta-Trait, Package Template, DeepFjig. Here we synthesize the main ideas of those very different designs into a nominally typed minimalist language, natural to OO programmers. By making our type system distinguish between code-use and code-reuse we can separate inheritance and subtyping, while avoiding redundant abstract declarations required in Delta-Trait and DeepFjig. At the same time self construction, binary methods and recursive types are also supported. Moreover, we provide a novel and elegant solution to uniformly handle behavior and state within trait composition. These ideas have been implemented in the 42 language, that supports all the examples we show in the paper.

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) **inherits** `B` code; and **create 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 separing subtyping from subclassing [11]. 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. [11], are classes with *binary methods* [10].
- **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 [17] 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 *this-leaking problem*, illustrated by the following (Java) code:

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

Method `A.ma` passes **`this`** as `A` to `Util.m`. 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 `A.ma` inside `B.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`.

As a though experiment, imagine that Java code-reuse (extends keyword) was not introducing subtyping: then an invocation of `B.mb` would result in a run-time type error. 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`.

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 in languages like Java classes do too many things at once. In particular they act both as units of *use* and *reuse*: classes can be *use*d as type and instantiated; classes can also be subclassed to provide *reuse* of code.

What the this-leaking problem shows is that adopting a more flexible nominally typed OO model where subclassing does not imply subtyping is not trivial, and a more substantial change in the language design is necessary. We are aware of at least 3 independently designed research languages that address this limitation

- Delta-Trait (DT) [6, 8, 7]: Each constructs has one single responsibilty: classes instantiate objects, interfaces induce types, records express state and traits are unit of reuse.
- Package Templates (PT) [15, 3, 2]: An extension of (full) Java where new packages can be "syntetized" by mixing and integrating code templates. As an extension of java, PT allows but not requires separation of inheritance and subtyping.
- DeepFjig [12, 22, 16]: A module composition language where the main idea is that nested classes with the same name are recursivelly composed.

This paper aims at showing a simple language design, called 42_μ, 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 42_μ there are two separate concepts: classes and traits [21]. Classes are meant for code use, and cannot be used for reuse. In some sense classes in 42_μ are like final classes in Java. Traits are meant for code reuse and multiple traits can be composed to form a class

[3] C++ allows to "extends privately", but it is a limitation over subtyping visibility, not over subtyping itself. The former example would be accepted even if `B` was to "privately extends" `A`

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 turn brings several benefits in terms of flexibility and code reuse. In summary, our contributions are:

- We identify the **this-leaking** problem and argue why it makes the separation of subclassing and subtyping difficult.
- We synthesize the key ideas of previous designs solving the this-leaking problemin into **a novel and simple language design**. This language is the logic core of the language 42, and all the examples in the paper can be directly encoded as valid 42 programs.
- We illustrate how the new design **improves both code use and code reuse**.
- We propose **a clean and elegant approach to handle of state** in a trait based language.

2 The Design of 42_μ: Separating Use and Reuse

This section presents the overview of 42_μ, and it illustrates the key ideas of its design. In particular we illustrate how to separate code use and code reuse, and how 42_μ solves the this-leaking problem.

2.1 Classes in 42_μ: A mechanism for code Use

The concept of a class in 42_μ provides a mechanism for code-use only. This means that there is actually no subclassing: 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 42_μ. Consider the example in Section 1:

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

Classes in 42_μ use a slightly different declaration style compared to Java: there is no **`class`** keyword, and a colon separates the class name (which must always start with an uppercase letter) and the class implementation, which is used to specify the definitions of the class. In our example, 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.

The 42_μ 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. This may seem like a severe restriction, but 42_μ has a different mechanism for *code-reuse* that is more appropriate when *code-reuse* is intended.

2.2 Traits in 42_μ: A mechanim of code Reuse

Unlike classes, traits in 42_μ 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 1 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`. A Java programmer may then try to write:

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

But this does not work either: `ta` is not a type in the first place, since it is a (lowercase) trait name. Indeed 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 42_μ.

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

```
IA:{interface method int ma()}//interface with abstract method
Utils:{class method int m(IA a)/*method body here*/}
ta:{implements IA //This line is the core of the solution
    method int ma() Utils.m(this) }
A:Use ta
```

This code works: `Utils` relies on interface `IA` and the trait `ta` implements `IA`. `ta` is well typed: independently of what class name(s) will be associated to its code, we know that such class(es) will implement `IA`. Therefore, while typechecking `Utils.m(this)` we can assume **`this`**`<: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`.

Later, in Section 5 we will provide more details on the type system. For now notice that in the former example the code is correct even if no method called `ma` is explicitly declared. DeepFJig and DT would require instead to explicitly declare an abstract `ma`:

```
B:Use ta, { method int ma()//not required by us
    method int mb(){return this.ma();} }
```

The idea is that such method is imported from trait `ta`, exactly as in the Java equivalent

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

where method `ma` is imported from `A`. This concept is natural for a Java programmer, but was not supported in previous work [**?**,12]. Those works require all dependencies in code literals to be explicitly declared, so that the code literal is completely self-contained. However, this results in many spurious abstract method declarations.

Semantic of Use Albeit alternative semantic models for traits [21] 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 is correct, and **in the resulting code there is no mention of the trait `ta`**. In some sense, all the information about code-reuse/inheritance is just a private implementation detail of `A` and `B`; while subtyping is part of the class interface.

3 Improving Use

To illustrates how 42_μ *improves the use* of classes we model a simplified version of Set and Bag collections first in Java, and then in 42_μ. The main benefit of the 42_μ solution is that we can get reuse without introducing subtyping between Bags and Sets. This improves the use of Bags and eliminates logical errors arizing from incorrect subtyping relations that are allowed in the Java solution.

3.1 Sets and Bags in Java

An iconic example on why connecting inheritance/code reuse and subtpying is problematic is provided by LaLonde [17]. A reasonable implementation for a `Set` is easy to extend into a `Bag` by keeping track of how many times an element occurs. We just need to add some state and override a few methods. For example in Java one could have:

```
class Set {...//usual hashmap implementation
  private Elem[] hashMap;
  void put(Elem e) /*body*/
  boolean isIn(Elem e)/*body*/}
class Bag extends Set{...//for each element in the hash map,
  private int[] countMap;// keep track of how many times is there
  @Override void put(Elem e)/*body*/
  int howManyTimes(Elem e)/*body*/}
```

Coding `Bag` in this way avoids a lot of code duplication, but *we induced unintented subtyping*! Since subclassing implies subtyping, our code break Liskov substitution principle (LSP) [18]: not all bags are sets! Indeed, the following is allowed:

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

A (broken) attempt to fix the Problem in Java. One could *retroactivelly* fix this problem by introducing `AbstractSetOrBag` and making both `Bag` and `Set` inherit from it:

```
abstract class AbstractSetOrBag{/*old set code goes here*/}
class Set extends AbstractSetOrBag{} //empty body
class Bag extends AbstractSetOrBag{/*old bag code goes here*/}
...
//AbstractSetOrBag type not designed to be used.
AbstractSetOrBag unexpected=new Bag();
```

This looks unnatural, since `Set` would extend `AbstractSetOrBag` without adding anything, and we would be surprised to find a use of the type `AbstractSetOrBag`. Worst, if we are to constantly apply this mentalty, we would introduce a very high number of abstract classes that are not supposed to be used as types. Those classes would clutter the public interface of our classes and our code project as a whole. In our example the information `Set<:AbstractSetOrBag` would be present in the public interface of the class `Set`, but it is not needed to use the class properly!

Moreover, the original problem is not really solved, but only moved further away. For example, one day we may need bags that can only store up to 5 copies of the same element. We are now at the starting point again:

– either we insert `class Bag5 extends Bag` and we break LSP;

- or we duplicate the code of the `Bag` implementation with minimal adjustments in **`class`** `Bag5` **`extends`** `AbstractSetOrBag`;
- or we introduce a **`abstract class`** `BagN` **`extends`** `AbstractSetOrBag` and **`class`** `Bag5` **`extends`** `BagN` and we modify `Bag` so that **`class`** `Bag` **`extends`** `BagN`. Note that this last solution is changing the public interface of the formerly released `Bag` class, and this may even break retro-compatibility (if a client program was using reflection, for example).

3.2 Sets and Bags in 42_μ

Instead, in 42_μ, if we was to originally declare

```
Set:{/*set implementation*/}
```

Then our code would be impossible to reuse in the first place for any user of our library. We consider this an advantage, since unintended code reuse runs into underdocumented behaviour nearly all the time[4]! If the designer of the set class wishes to make it reusable, it can do it explicitly by providing a set trait:

```
set:{/*set implementation*/}
Set:Use set
```

Since `set` can never be used as a type, there is no reason to give it a **fancy-future-aware** name like `AbstractSetOrBag`. When bag will be added, the code will look either like

```
set:{/*set implementation*/}
Set:Use set
Bag: Use set, {/*bag implementation*/}
```

or

```
set:{/*set implementation*/}
Set:Use set
bag: Use set, {/*bag implementation*/}
Bag: Use bag
```

Notice how, thanks to flattening, the resulting code for `Bag` is identical in both versions, and as shown in Section 2, there is no trace of trait `bag`. Thus if we are the developers of bags, we can temporarly go for the first version, and move when needed to the second without adding new undesired complexity for our old clients.

4 Improving Reuse

42_μ allows reuse even when subtyping is impossible. In 42_μ traits do not induce a new (externally visible) type. However, locally in the trait the programmer can use the special self-type **`This`** in order to denote the type of **`this`**. That is: the program is agnostic to what the **`This`** type is going to be, so it can be later assigned to any (or many) classes. The idea is that during flattening, **`This`** will be replaced with the actual class name. In this way, 42_μ allows reuse even when subtyping is impossible, that is, when a method parameter has type **`This`** (also called *binary methods* [10]). This type of situations is the primary motivator for previous work aiming at separating inheritance from subtyping. Leveraging on the **`This`** type, we can also provide self-instantiation (trait methods can create instances of the class using them) and smootly integrate state and traits: a challenging problem that has limited the flexibility of traits and reuse in the past.

[4] See "Design and document for inheritance or else prohibit it"[9]: the self use of public methods is rarelly documented, thus is hard to understand the effects of overriding a library metod.

4.1 Managing State

To illustrate how 42_μ improves reuse, we will show a novel approach to deal with *state* in traits. The idea of summing pieces of code is very elegant, and has proven very successfull in module composition languages [1] and several trait models [21, 5, 8, 16]. However the research community is struggling to make it work with object state (constructor and fields) while achieving the following goals:

(1) managing fields in a way that borrows the elegance of summing methods; (2) actually initialize objects, leaving no null fields; (3) making it easy to add new fields; (4) allowing a class to create instances of itself.

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 class has syntax richer that the trait one, allowing declaration for fields and constructors. This approach is very powerful as illustrated by Wang et al. [23].

Immutable Points Consider, for example, two simple traits that deal with *immutable point* objects. That is, points in the cartesian plane (with coordinates `x` and `y`). The first trait provides a *binary method* that sums the point object with another point to return a new point. The second trait provides a similar operation that does multiplication instead.

```
pointSum: { method int x()  method int y()//getters
  class method This of(int x,int y)//factory method
  method This sum(This that)
    This.of(this.x()+that.x(),this.y()+that.y())//self instantiation
  }
pointMul: { method int x() method int y()//repeating getters
  class method This of(int x,int y)//repeating factory
  method This mul(This that)
    This.of(this.x()*that.x(),this.y()*that.y())
  }
```

As we can see, all the state operations (the getters for the `x` and `y` coordinates) are represented as **abstract** methods. Notice the abstract **`class method This`** `of(..)` which acts as a constructor for points: A class method is similar to a **`static`** method in Java but can be abstract. As for instance methods, they are late bound: flattening can provide an implementation for them. abstract class method are very similar to the original concept of member functions in the module composition setting.

4.2 A First Attempt at Composition

According to the general ideas about trait composition presented in Section 2, a first attempt at composing the two traits providing two different operations on points is:

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

However this fails since methods `x`,`y` and `of` are still abstract! Instead a user could 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 }
}
```

This approach works, and it as some advantages, but also some disadvantages:

- **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 the methods returning **`This`** by forwarding a call to the constructor.
- **Disadvantages:** Writing those obvious definitions to close the state/fixpoint in the class with the constructors and fields and getter/setters and factories can be quite tedious. Previous work shows that such code can be automatically generated [23].

4.3 The 42_μ approach to State

In our model we go one step further: there is no need to generate code, or to explicitly write down constructors and fields. In fact in 42_μ there is not even syntax for those constructs! The idea is that any class that "can" be completed in the obvious way is *a complete "coherent" class*. In most other languages, a class is abstract if it has abstract methods. Instead, we call a class abstract only when the set of abstract methods is not coherent. That is, the unimplemented methods cannot not be automatically recognized as factory, getters and setters. Methods recognized as factory, getters and setters are called *abstract state operations*.

Coherent classes A more detailed definition of coherent classes is given next:

- a class with no abstract methods is coherent (just like Java `Math`, for example). Such classes are useful 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 interpreted 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: it creates a new instance that is like **`this`** but where field `x` has now `that` value.
- **`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. The work by Wang et al. [23] explores a particular set of such abstract state operations.

Points in 42_μ*:* In 42_μ and with our approach to handle the state, `pointSum` and `pointMul` can be directly composed:

```
PointAlgebra:Use pointSum,pointMul
```

Note how we can declare the methods independently and compose the result as we wish.

Improved solution In this first attempt, we repeated the abstract methods `x`,`y` and `of`. Moreover, in addition to `sum` and `mul` we may want many operations over points. It is possible to improve reuse and not repeat such abstract definitions by abstracting the common abstract definitions into a trait `p`:

```
p: { method int x() method int y()
  class method This of(int x,int y)
  }
pointSum:Use p, { method This sum(This that)
    This.of(this.x()+that.x(),this.y()+that.y())
  }
pointMul:Use p, { method This mul(This that)
    This.of(this.x()*that.x(),this.y()*that.y())
  }
pointDiv: ...
PointAlgebra:Use pointSum,pointMul,pointDiv,...
```

Now the code is fully modularized, and each trait handles exactly one method.

4.4 State Extensibility

Programmers may want to extend points with more state. For example they may want to add colors to the points. A first attempt at doing this would be:

```
colored:{ method Color color() }
Point:Use pointSum,colored //Failure: resulting class not coherent
```

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 support for overloading on different number of parameters. This is a reasonable solution, however the method `CPoint.sum` resets the color to red: we call the `of(int,int)` method, that now delegates to `of(int,int,Color)` by passing red as the default field value. What should be the behaviour in this case? If our abstract state supports withers, instead of writing `This.of(...)` we can use `this.withX(newX).withY(newY)` in order to preserve the color from **this**. This solution is still not satisfactory: this design ignores the color from `that`.

A better design If the point designer is designing for reuse and extensibility, then a better design would be the following:

```
p: { method int x() method int y() //getters
  method This withX(int that) method This withY(int that)//withers
  class method This of(int x,int y)
  method This merge(This that) //new method merge!
  }
pointSum:Use p, { method This sum(This that)
    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) //how to merge color handled here
    this.withColor(this.color().mix(that.color())
```

```
  }
CPoint:/*as before*/
```

This design allows merging colors, or any other kind of state we may want to add following this pattern.

Independent Extensibility Of course, quite frequentely there can be multiple independent extensions [24] that need to be composed. Lets suppose that we could have a notion of flavoured points as well. 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 [13].

In the following you can see how to mix colors and flavours. The syntax `_2merge` and `_3merge` call the version of merge as defined in the second/third element of **`Use`**.

```
p: {/*as before*/ }
pointSum:Use p, {/*as before*/ }
colored:{/*as before*/}
flavored:{method Flavor flavor() //very similar to colored
  method This withFlavor(Flavor that)
  method This merge(This that) //how to merge flavors handled here
    this.withFlavor(that.flavor())}//here we just inherit ''that'' flavor
FCPoint:Use pointSum,colored,flavored{
  class method This of(int x,int y)
    This.of(x,y,Color.of(/*red*/),Flavor.none())
  class method This of(int x, int y,Color color,Flavor flavor)
  //we can resolve the conflict about two implementations for merge
  //by proving our own implementation here
  method This merge(This that) this._2merge(that)._3merge(that)
  }
```

5 Intuitions on formalization

In this article we dedicate more space to examples and informal presentation and motivations; so we do not have space to provide a full formalizations. We will provide here some hints on how the formalization works.

In the following, we present a very simplified grammar:

TD	$::= t : L \mid t$ **`:Use`** $\overline{V}$	Trait Decl
CD	$::= C : L \mid C$ **`:Use`** $\overline{V}$	Class Decl
V	$::= t \mid L$	Code Value
L	$::=$ `{` **`interface`**$?$ **`implements`** $\overline{T}\ \overline{MD}$ `}`	Code Literal
T	$::= C$	types are class names
MD	$::=$ **`class`**$?$ **`method`** $T\ m$ `(` $\overline{T\ x}$ `)` $e?$	Method Decl
e	$::= x \mid T \mid e$ `.` m `(` $\overline{e}$ `)`	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 in this work we focus on the single operator **`Use`**, taking a set of code values: that is trait names t or literals L and composing them. This operation, sometimes called *sum*, is the simplest

and most elegant trait 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.

Code literals L can be marked as interfaces. That is, the interface keyword is inside the curly brakets, so an upper case name associated with an interface literal is a class-interface, while a lowercase one is a trait-interface. In our simple model, we consider an error trying to merge an interface with a non-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 [14].

Well-formedness Basic well formedness rules apply:

- 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,
- all methods in an interface are abstract, and there are no interface class methods.

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

We expect the type system to enforce:

- all the traits and classes have unique names in a program $\overline{D}$, and the special class name **This** is reserved,
- all used types correspond to class declarations in the program, or are **This**,
- 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

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 flattening completes creating a class definition, **This** will be replaced with such class name.

For the sake of simplicity, method bodies are just simple expressions e: they can be just variables, types and method calls. We need types as part of expressions in order to use them as receivers for class methods.

5.1 Remarks on Typing

Our typing discipline is what distinguishes our approach from a simple minded code composition macro [4] or a rigid module composition [1].

The are two core ideas: *1: 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.
2: code literals 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 [12, 7, 5] 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 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 one 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 (indirectly) 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.

5.2 Recursive types

OO languages 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 program 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. That is, In our approach typing and flattening interleaves. We assume our compilation process to stop as soon as an error arise. There are two main kinds of errors: Type errors (like method not found) or Composition errors (like summing two conflicting implementation for the same method). 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 to modularly ensure well typedness, we require `tb` to be well typed at this stage. If `tb` was not well typed a type 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 composition 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 without information about `B`. A type 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 it 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 an involved 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 flattening.

In the world of strongly typed languages we are 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 **only** the code 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.

6 Related Work

Literature on code reuse is too vast to let us do justice of it in fiew pages, Our work is inspired on Traits [13]; they are in turn insipired on module composition languages [1].

6.1 Inheritance and subtyping

We are aware of at least 3 independently designed research languages that address this limitation:Delta-Trait(DT)[**?**], Package Template(PT)[], DeepFjig[]. Leveraging on *traits*, in this work we aim to synthesizes the main ideas of those very different designs.

- DeepFjig+ **A simple uniform syntax for code literals** DeepFjig is best in this sense, since DT have separate syntax for class literals, trait literals and record literals. PT on the other side build on top of full Java, thus has a very involved syntax. Thanks to our novel represantation of state, our approach offers a much simpler and uniform syntax then all others, where evertyhing is basically just a method.
- DeepFjig- **Reusable code can not be "used", that is instantiated or used as a type.** This happens in DT and in PT, but not in DeepFjig. To allow reusable code to be directly usable, in DeepFjig classes introduce nominal types in an innatural way: the type of **`this`** is only **`This`** (sometimes called <>) and not the nominal type of its class: that is in DeepFjig `A:{ method A m()this}` is not well typed. This is because `B: Use A` flattens to `B:{ method A m()this}`, that is clearly not well typed. Looking to this example is clear why we need reusable code to be agnostic of its name. Then, either reusable code have no name (DT, PT and this work) or all code is reusable and usable, and all code need to be auckwardly agnostic of its name.

– PT+ **Requiring abstract signatures is a left over of module composition mindset.** DT and Deepfjig comes form a tradition of functional module composition, where modules are typed in isolation under an environments, and then the composition is performed. As we show in this work, this is ends up requiring verbose repetition of abstract signatures, that for hightly modularized code may end up composing most of the program. Simple Java (and thus PT, since it is an add on over Java) show us a better way: the meaning of names can be understood from the reuse context. The typing strategy of PT offers the same advantages of our typing model, but is more involved and indirect. This may be caused by the heavy task of integrating with full Java.
– PT-**Composition algebra.** The idea of using composition operators over atomic values as in an arithmetic expression is very powerfull, and makes easy to extend languages with more operators. Deepfjig and DT embrace this idea, while PT takes the traditional Java/C++ approach of using enhanced class/package declaration syntax. The typing strategy of PT also seams to be connected with this decision, so it would be hard to move their approach in a composition algebra setting.
– DT- **Naming your type, even if you have none yet.** Both Deepfjig and PT allows a class to refer to its name, abait is more unobvious in PT since you have to introduce both a package and a classes to express it. This allows to encode binary methods, express patterns like withers or fluent setters and to instantiate instances of the (future) class(es) using the reused code.
– DT+**Complete ontological separation between use and reuse** While all 3 works allows to separate inheritance and subtyping only DT properly enforce separation between use (classes and interfaces) and reuse (traits). This is because in DeepFjig all classes are both units of use and reuse (however, subtype is not induced) and PT imports all the complexity of Java, so abait is possible to separate use and reuse, the model have powerfull but unobvous implications where (conventional Java) **`extends`** and PT are used together

With the exception of those 3 lines of work, to the best of our understading other famous work in literature, like [20, 19] does no completly break the relation between inheritance and subtyping, but only prevent subtyping only where it would be unsound.

6.2 State and traits

The idea of abstract state operation has emerged from CJ [23], 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. Fields are just like another kind of (abstract) member, and two fields with identical type can be merged by sum/use; **`new`** `C()` can be used for all classes, and `init` methods 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 (used by [**?**]) the fields are declared but not initialized, and a canonical constrctor constructor (as in FJ) taking a value for each field and just initializing such field is automatically generated in the resulting class.
  It is easy to add fields, however this model is associative but not commutative: composition order influence field order, and thus the construtor signature. Self construction is not possible since the signature of the constuctor change during composition.
– Constructor can be composed if they offers the same exact parameters: In this approach (used by DeepFJig) 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: composition order only influence execution order of side effects during construction. However constructors composition requires identical constructor signature: this hamper reuse, and if a field is added, its initial value need to be somehow syntetized from the constructor parameters.

6.3 Tablular comparision of many approaches

In this table we show if some approach or part-there-of support certain critical features: Direct instantation (as in `new C()`), Self instantiation (as in `new This()`), Unit of use, Unit of reuse, Introduce type and if the Induced type is the type of `this`, support for binary methods, does inheritance induce subtype?, is code required to be well-typed before being inherited /imported in a new context? is code required to be well-typed before composed with other code?
We use Y and X to mean yes and no, and we use "-" were the question is not really applicable to the current approach. For example the original trait model was untyped, so typing questions makes little sense.

	direct instantation	self instantiation	unit of use	unit of reuse	introduce type	induced type is this type	binary methods	inheritance induce subtype	well-typed before imported	well-typed before composed
java/scala class	Y	X	Y	Y	Y	Y	X	Y	Y	X
java8 interface	X	X	X	Y	Y	Y	X	Y	Y	X
scala trait	X	X	X	Y	Y	Y	X	Y	X	X
original trait	X	X	X	Y	-	-	X	X	-	-
Ferruccio trait	X	X	X	Y	X	-	X	X	Y	Y
42_μ trait	X	Y	X	Y	X	-	Y	X	Y	X
42_μ class	Y	Y	Y	X	Y	Y	Y	-	Y	X
module composition	-	-	Y	Y	-	-	-	-	Y	Y
deepFJig class	Y	Y	Y	Y	Y	X	Y	X	Y	Y
package template	X	Y	X	Y	X	-	X	X	Y	X

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.

MARCO: Cite some work of bruceTHisType and show how he also fails to separe reuse and use

7 Conclusions, extensions and practical applications

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

The model presented here is very easy to extend: For example nested classes can be easly added,giving us enough power to easly solve the expression problem as in [12]. More composition operators can be added in addition to **`Use`**. In particular all the sophisticated operators of [12] could be added. Indeed we can add any operator respecting following criteria:

- As for **`Use`**, the operator 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 literals, 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.

Our simplified model represent the conceptual core of a novel full blown programming language (`http://L42.is`), 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 `github.com/ElvisResearchGroup/L42/tree/master/Main/formal`. L42 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. L42 also 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.

L42 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.

References

1. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of functional programming*, 12(02):91–132, 2002.
2. Eyvind W. Axelsen and Stein Krogdahl. Package templates: a definition by semantics-preserving source-to-source transformations to efficient java code. In Klaus Ostermann and Walter Binder, editors, *Generative Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012*, pages 50–59, 2012.
3. Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. Challenges in the design of the package template mechanism. *Trans. Aspect-Oriented Software Development*, 9:268–305, 2012.
4. Alan Bawden et al. Quasiquotation in lisp. In *PEPM*, pages 4–12. Citeseer, 1999.
5. Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. *Stateful Traits*, pages 66–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

6. Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing software product lines using traits. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2096–2102. ACM, 2010.
7. Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming*, 97, Part 3:282 – 308, 2015.
8. Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Science of Computer Programming*, 78(5):521 – 541, 2013.
9. Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. Prentice Hall PTR, 2 edition, 2008.
10. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3), 1996.
11. William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, 1990.
12. Andrea Corradi, Marco Servetto, and Elena Zucca. Deepfjig - modular composition of nested classes. *Journal of Object Technology*, 11(2):1: 1–42, 2012.
13. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006.
14. Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.
15. Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. Exploring the use of package templates for flexible re-use of collections of related classes. *Journal of Object Technology*, 8(7):59–85, 2009.
16. Giovanni Lagorio, Marco Servetto, and Elena Zucca. Featherweight jigsaw - replacing inheritance by composition in java-like languages. *Inf. Comput.*, 214:86–111, 2012.
17. Wilf LaLonde and John Pugh. Smalltalk: Subclassing subtyping is-a. *J. Object Oriented Program.*, 3(5):57–62, January 1991.
18. Robert C Martin. Design principles and design patterns. *Object Mentor*, 1(34), 2000.
19. Nathaniel Nystrom, Xin Qi, and Andrew C Myers. J&: nested intersection for scalable software composition. In *ACM SIGPLAN Notices*, volume 41, pages 21–36. ACM, 2006.
20. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.
21. Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP 2003 — Object-Oriented Programming, 17th European Conference*, volume 2743 of *Lecture Notes in Computer Science*, 2003.
22. Marco Servetto and Elena Zucca. A meta-circular language for active libraries. *Science of Computer Programming*, 95:219–253, 2014.
23. Yanlin Wang, Haoyuan Zhang, Bruno C d S Oliveira, and Marco Servetto. Classless java. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pages 14–24. ACM, 2016.
24. Mathhias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *Foundations of Object-Oriented Languages (FOOL)*, January 2005.