

# Separating Use and Reuse to Improve Both

Hrshikesh Arora<sup>a</sup>, Marco Servetto<sup>a</sup>, and Bruno C. d. S. Oliveira<sup>b</sup>

<sup>a</sup> Victoria University of Wellington

<sup>b</sup> The University of Hong Kong

## Abstract

*Context:* Trait composition has inspired new research in the area of code reuse for object oriented (OO) languages. One of the main advantages of this kind of composition is that it makes possible to separate subtyping from subclassing; which is good for code-reuse, design and reasoning [15]. However, handling of state within traits is difficult, verbose or inelegant.

*Inquiry:* We identify the *this-leaking problem* as the fundamental limitation that prevents the separation of subtyping from subclassing in conventional OO languages. We explain that the concept of trait composition addresses this problem, by distinguishing code designed for use (as a type) from code designed for reuse (i.e. inherited). We are aware of at least 3 concrete independently designed research languages following this methodology: TraitRecordJ [6], Package Templates [26] and DeepFJig [16].

*Approach:* In this paper, we design  $42_{\mu}$ , a new language, where we improve use and reuse and support the **This** type and family polymorphism by distinguishing code designed for use from code designed for reuse. In this way  $42_{\mu}$  synthesise the 3 approaches above, and improves them with *abstract state operations*: a new elegant way to handle state composition in trait based languages.

*Knowledge and Grounding:* Using case studies, we show that  $42_{\mu}$ 's model of traits with abstract state operations is more usable and compact than prior work. We formalise our work and prove that type errors cannot arise from composing well typed code.

*Importance:* This work is the logical core of the programming language 42. This shows that the ideas presented in this paper can be applicable to a full general purpose language. This form of composition is very flexible and could be used in many new languages.

ACM CCS 2012

- **Theory of computation** → **Formalism**;
- **Software and its engineering** → **Object oriented languages; Designing software; Formal language definitions**;

**Keywords** trait composition, subtyping, code reuse

## The Art, Science, and Engineering of Programming

Perspective The Theoretical Science of Programming

Area of Submission Type systems



© Hrshikesh Arora, Marco Servetto, and Bruno C. d. S. Oliveira  
This work is licensed under a "CC BY 4.0" license.  
Submitted to *The Art, Science, and Engineering of Programming*.

## Separating Use and Reuse to Improve Both

### 1 Introduction

In Java, C++, Scala and C#, subclassing implies subtyping. A Java subclass declaration, such as `class A extends B {}` does two things at the same time: it **inherits** code from B; and it creates a subtype of B. Therefore a subclass must *always* be a subtype of the extended class. Such design choice where subclassing implies subtyping is not universally accepted. Historically, there has been a lot of focus on separating subtyping from subclassing [15]. This separation 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:** Situations where inheritance is desirable are prevented by the enforced subtyping relation. A well-known example are the so-called *binary methods* [11, 15]. For example, consider a class `Point` with a method `Point sum(Point o){return new Point(x+o.x,y+o.y);}`. Can we reuse the `Point` code so that `ColorPoint.sum` would take and return a `ColorPoint`? In Java/C# declaring `class ColorPoint extends Point{..}` would result in `sum` still taking a `Point` and returning a `Point`. Moreover, manually redeclaring a `ColorPoint sum(ColorPoint that)` would just induce overloading, not overriding. In this case we would like to have inheritance, but we cannot have (sound) subtyping.
- **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 sound. A typical example [29] is *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.

Structural typing [15] may deal with the first situation, 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: an explicit subtyping relation must be signalled by the programmer. Thus if subtyping is not desired, the programmer can simply **not** declare a subtyping relationship.

While there is no problem in subtyping without subclassing, in most nominal OO languages subclassing fundamentally implies subtyping. This is because of what we call the *this-leaking problem*, illustrated by the following (Java) code, where method `A.ma` passes **this** as `A` to `Utils.m`. This code is correct, and there is no subtyping/subclassing.

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

Now, lets add a class B

```
1 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 `Utils.m` with an instance of B. However, *this can be correct only if B is a subtype of A*.

Suppose Java code-reuse (the **extends** keyword) did not introduce subtyping<sup>1</sup>: 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 `B.mb`), it will have a type that is incompatible with the expected argument of type `A`. Therefore, 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.

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: a more substantial change in the language design is necessary. 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 *used* as types and can be instantiated; classes can also be subclassed to provide *reuse* of code. We are aware of at least 3 independently designed research languages that address the *this-leaking problem*:

- In **TraitRecordJ (TR)** [6, 7, 8] each construct has a single responsibility: classes instantiate objects, interfaces induce types, records express state, and traits are reuse units.
- **Package Templates (PT)** [2, 3, 26]: an extension of (full) Java where new packages can be “synthesized” by mixing and integrating code templates. Such “synthesized” packages can be used for code reuse without inducing subtyping.
- **DeepFJig(DJ)** [16, 28, 37] is a module composition language where nested classes with the same name are recursively composed.

This paper shows a simple language design, called  $42_{\mu}$ , addressing the *this-leaking problem* and decoupling subtyping from inheritance. We build on traits to distinguish code designed for *use* from code designed for *reuse*. We synthesize and simplify the best ideas from those 3 very different designs, and couple them with an elegant novel approach to state and self instantiation in traits that avoids the complexities and redundancies introduced by fields and their initialisation.

In  $42_{\mu}$ , there are two separate concepts: classes and traits [18]. Classes are meant for code *use*, and cannot be inherited/extended. Classes in  $42_{\mu}$  are like final classes in Java, and can be used as types and as object factories. Traits are meant for code *reuse* only: multiple traits can be composed to form a class. However, traits cannot be instantiated or used as types. This allows fine-grained control of subtyping while handling examples like `Set/Bag`.

In  $42_{\mu}$ , as in many module composition languages [1], all methods can be abstract, including static ones. Moreover, module composition can be used to make an already implemented method abstract. Thus, as for dynamic dispatch, the behaviour of a method call is never set in stone. We will show how in  $42_{\mu}$ , state is induced by an implicit fixpoint operation over abstract methods, where an abstract static method can

<sup>1</sup> C++ allows “extending privately”; this is not what we mean by not introducing subtyping: in C++ it is a limitation over subtyping visibility, not over subtyping itself. Indeed, the former example would be *accepted* even if `B` were to “privately extends” `A`

## Separating Use and Reuse to Improve Both

perform the role of a constructor. This allows handling examples like `Point/ColorPoint` in a natural way, without requiring code duplication.

Our design brings several benefits. In particular, Family Polymorphism [20] is radically simpler to support soundly. This is already clear in the 3 lines of research above, and is even more outstanding in the clean  $42_\mu$  model.

We first focus on an example-driven presentation to illustrate how to improve use and reuse. In Appendix A, we then provide a compact formalization. The hard technical aspects of the semantics have been studied in previous work [2, 3, 6, 7, 8, 16, 26, 28, 37]; the design of  $42_\mu$  synthesizes some of those concepts. The design ideas have been implemented in the full  $42$  language, which supports all the examples we show in the paper, and is available at: <http://l42.is>. Work on  $42$  is now slowly reaching maturity after about 5 years of intense research and development. The current implementation is now robust enough to create realistic medium sized programs running on the JVM, and the standard library consists of over 10000 lines of  $42$  code.

In summary, our contributions are:

- We identify the this-leaking problem, that makes separating inheritance and sub-typing difficult.
- We synthesize the key ideas of previous designs that solve the this-leaking problem into a novel and minimalistic language design. This language is the core logic of the language  $42$ , and all the examples in this paper can be encoded as valid  $42$  programs. This design improves both code use and code reuse.
- We propose a clean and elegant approach to the handling of state in a trait based language.
- We illustrate how  $42_\mu$ , extended with nested classes, enables a powerful (but at the same time simple) form of family polymorphism.
- We show the simplicity of our approach by providing a compact 1 page formalization (in Appendix A).
- We perform 3 case studies, comparing our work with other approaches, and we collect clear data showing that we can express the same examples in a cleaner and more modular manner.

## 2 The Design of $42_\mu$ : Separating Use and Reuse

### 2.1 Classes in $42_\mu$ : a mechanism for code use

Consider the example of Section 1 rewritten in  $42_\mu$ , introducing classes `Utils` and `A`:

```
1 A={ method int ma(){ return Utils.m(this); } }
2 Utils={ static method int m(A a){ return ..; } }
```

Classes in  $42_\mu$  use a different declaration style compared to Java: there is no `class` keyword, and an equals sign 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 associated with the class is `{ method int ma(){return Utils.m(this);}}`

and it contains the method `ma()`. In the  $42_\mu$  code above, there is no way to add a class `B` reusing the code of `A`: class `A` (uppercase) is designed for code *use* and not *reuse*. Indeed, a noticeable difference with Java is the absence of the **extends** keyword.  $42_\mu$  classes are roughly equivalent to final classes in Java. This means that there is actually no subclassing. Thus, unlike the Java code, introducing a subclass `B` is not possible. This may seem like a severe restriction, but  $42_\mu$  has a different mechanism for *code-reuse* that is more appropriate when *code-reuse* is intended.

## 2.2 Traits in $42_\mu$ : a mechanism for code reuse

Traits in  $42_\mu$  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 (even syntactically they can not be used as types).

```
1 Utils={ static method int m(A a){return ...} }
2 ta={ method int ma(){return Utils.m(this);}} //type error
3 A=Use ta
```

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 reusing the code from the trait `ta`, introduced by the keyword **Use**. Note that **Use** expressions cannot contain class names: only trait names are allowed. *Referring to a trait is the only way to induce code reuse.*

The crucial point is the call `Utils.m(this)` inside trait `ta`: the corresponding call in the Java code is correct since Java guarantees that such occurrence of **this** will be a subtype of `A` everywhere it is reused. In  $42_\mu$  the type of **this** in `ta` has no relationship to the type `A`; thus the code `Utils.m(this)` is illtyped.

The following second attempt would not work either:

```
1 Utils={ static method int m(ta a){return ...}} //syntax error
2 ta={ method int ma(){return Utils.m(this);}}
3 A=Use ta
```

`ta` is not a type in the first place, since it is a (lowercase) trait name. Indeed, trait names can only be used in **Use** expressions, and thus they can not appear in method bodies or type annotations. In this way, the code of a trait can stay agnostic of its name. This is one of the key design decisions in  $42_\mu$ : traits can be reused in multiple places, and their code can be seen under multiple types. In  $42_\mu$ , *interfaces are the only way to obtain subtyping*. As shown in the code below, interfaces are special kinds of code literals, where all the methods are abstract. Thus, to model the original Java example, we need an interface capturing the commonalities between `A` and `B`:

```
1 IA={interface method int ma()} //interface with abstract method
2 Utils={static method int m(IA a){return ...} }
3 ta={implements IA //This line is the core of the solution
4   method int ma(){return Utils.m(this);}}
5 A=Use ta
```

This code works: `Utils` relies on interface `IA` and the trait `ta` implements `IA`. Any class reusing `ta` will contain the code of `ta`, including the **implements** `IA` subtyping declaration; thus any class reusing `ta` will be a subtype of `IA`. Therefore, while typechecking `Utils.m(this)` we can assume `this<:IA`. It is also possible to add a class `B` as follows:

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

## Separating Use and Reuse to Improve Both

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

Later, in Appendix A we will provide the type system. Here notice that the former declaration of  $B$  is correct even if no method called  $ma$  is explicitly declared. DJ and TR would instead require explicitly declaring an abstract  $ma$  method:

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

In  $42_{\mu}$ , methods are directly accessible from  $ta$ , exactly as in the Java equivalent

```
1 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 [8, 16]. Those works require all dependencies in code literals to be explicitly declared, so that the code literal is self-contained; in this way it can be typed in isolation before flattening. However, this results in many redundant abstract method declarations.

**Semantics of Use:** The semantics of traits is defined with flattening, which is simple to formalize and understand. However, if implemented naively, flattening may cause a lot of bytecode duplication. The ‘delegation semantics’ [27], is a proposed alternative semantic model for traits that is observationally equivalent to flattening but does not require bytecode duplication. The formalism presented here will rely on simple flattening, but we expect the techniques of [27] would be useful to produce an efficient implementation in term of bytecode space. With the flattening semantics  $A$  and  $B$  are equivalent to the inlined code of all used traits.

```
1 A=Use ta
2 B=Use ta, { method int mb(){return this.ma();} }
3 //equivalent to
4 A={implements IA method int ma(){return Utils.m(this);}}
5 B={implements IA
6   method int ma(){return Utils.m(this);}
7   method int mb(){return this.ma();} }
```

*In the resulting code, there is no mention of the trait  $ta$ .* Information about code-reuse/inheritance is a private implementation detail of  $A$  and  $B$ ; while subtyping is part of the class interface. This position has been defended by Bracha [10]: the choice of inheriting behaviour should be in the hands of the programmer; if a method implementation is not appropriate, such method can be overridden. If too many methods do not provide an appropriate behaviour, inheriting code from another location or implementing the behaviour from scratch may also be considered. This should not impact the interface exposed to the user, otherwise the programmer may be unable to change their implementation decisions at a later time. In summary, to leak **this** in  $42_{\mu}$ , either code reuse is disallowed, or an appropriate interface ( $IA$  in this case) must be implemented. We believe the code with  $IA$  better transmits programmer intention. Some readers may instead see requiring  $IA$  as a cost of our approach. Even from this point of view, such cost is counter balanced by the very natural and simple support for code reuse, ‘**This**’ type and (in the extensions with nested classes seen later) family polymorphism. The syntactic cost of introducing new names can be reduced with some syntactic sugar, as explained in Appendix B.1.

### 3 Improving Use

To illustrate how  $42_{\mu}$  improves the use of classes, we model a simplified version of Set and Bag collections first in Java, and then in  $42_{\mu}$ . The benefit of  $42_{\mu}$  is that we get reuse without introducing subtyping between Bags and Sets. As shown below, this improves the use of Bags by eliminating logical errors arising from incorrect subtyping relations that are allowed in the Java solution.

#### 3.1 Sets and Bags in Java: the need for code reuse without subtyping

An iconic example on why connecting inheritance/code reuse and subtyping is problematic is provided by LaLonde [29]. 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 add some state and override a few methods. For example in Java one could have:

```

1 class Set {..//usual hashmap implementation
2   private Elem[] hashMap;
3   void put(Elem e){..}
4   boolean isIn(Elem e){..}}
5 class Bag extends Set{ ..//for each element in the hash map,
6   private int[] countMap;// keep track of how many occurrences are in the bag
7   @Override void put(Elem e){..}
8   int howManyTimes(Elem e){..}}
```

Coding Bag in this way avoids a lot of code duplication, but we induced unintended subtyping! Since subclassing implies subtyping, our code breaks the Liskov substitution principle (LSP) [30]: not all bags are sets!<sup>2</sup> Indeed, the following is allowed:

```

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

This encumbers the programmer: to avoid conceptual errors that are not captured by the type system, they have to *use* Bag very carefully.

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

```

1 abstract class AbstractSetOrBag {/*old set code goes here*/}
2 class Set extends AbstractSetOrBag {} //empty body
3 class Bag extends AbstractSetOrBag {/*old bag code goes here*/}
4 ...
5 //AbstractSetOrBag type not designed to be used.
6 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 mentally, 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 the project as a whole. A *useable* API should provide only the information relevant to the client. In our example, the information `Set<:AbstractSetOrBag` would be present in the public interface of the class Set, but

<sup>2</sup> The LSP is often broken in real programs because of the need of inheritance: the LSP allows only refinement not extension. Traits provide extension without breaking the LSP.

## Separating Use and Reuse to Improve Both

such information 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 the LSP;
- or we duplicate the code of the `Bag` implementation with minimal adjustments in `class Bag5 extends AbstractSetOrBag`;
- or we introduce an `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 backwards-compatibility (if a client program was using reflection, for example).

### 3.2 Sets and Bags in 42<sub>μ</sub>

Instead, in 42<sub>μ</sub>, if we were to originally declare

```
1 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 under-documented behaviour nearly all the time!<sup>3</sup> If the designer of the `Set` class wishes to make it reusable, they can do it explicitly by providing a `set` trait:

```
1 set={/*set implementation*/}
2 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` is added, the code will look like

```
1 set={/*set implementation*/}
2 Set=Use set
3 Bag=Use set, {/*bag implementation*/}
```

OR

```
1 set={/*set implementation*/}
2 Set=Use set
3 bag=Use set, {/*bag implementation*/}
4 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` at run time. Thus if we are the developers of bags, we can temporarily go for the first version. Then, when for example we need to add `Bag5` as discussed before, we can introduce the `bag` trait without adding new undesired complexity for our old clients.

<sup>3</sup> See “Design and document for inheritance or else prohibit it” [9]: the self use of public methods is rarely documented, thus is hard to understand the effects of overriding a library method.



## 4 Improving Reuse

To illustrate how  $42_\mu$  improves reuse, we show a novel approach to smoothly integrating state and traits: a challenging problem that has limited the flexibility of traits and reuse in the past. The idea of flattening is elegant and successful in module composition languages [1] and several trait models [5, 8, 18, 28]. Flattening is elegant in these two settings since traits (or modules) only have one kind of member: methods (or functions). In this way flattening is defined as simply collecting all members from all used traits (or composed modules), where methods with same name and type signature are summed into a single one. At most one of those summed methods can have a body, which will be propagated into the result. However the research community is struggling to make it work with object state (constructors and fields) while achieving the following goals:

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

An in-depth discussion on how such goals are difficult to achieve and how they have been challenged in the existing literature is available in Section 7.3.

### 4.1 State of the art

We first present the state of the art solution: traits have only methods but classes also have fields and constructors. The idea is that the trait code just uses getter/setters/factories, while leaving classes to finally define the fields/constructors. That is, in this state of the art solution, classes have a richer syntax than traits, allowing declaration for fields and constructors.

**Points:** Consider two traits dealing with *point* objects with coordinates  $x$  and  $y$ .

```

1 //idealized state of the art trait language, not 42
2 pointSum= { method int x() method int y()//getters
3   static method This of(int x,int y)//factory method
4   method This sum(This that){//sum code
5     return This.of(this.x()+that.x(),this.y()+that.y());//self instantiation
6   }}
7 pointMul= { method int x() method int y()//repeating getters
8   static method This of(int x,int y)//repeating factory
9   method This mul(This that){//multiplication code
10    return This.of(this.x()*that.x(),this.y()*that.y());
11  }}
```

The first trait provides a *binary method* that adds the point object to another point to return a new point. The second trait provides multiplication. In this code all the operations dealing with state are represented as *abstract methods*. Notice the abstract `static method This of(...)` which acts as a factory/constructor for points. As for instance methods, static methods are late bound: flattening can provide an implementation for them. Thus, in  $42_\mu$  they can be abstract, and abstract static methods are similar to the concept of member functions in the module composition setting [1].

## Separating Use and Reuse to Improve Both

Following the traditional model of traits and classes common in literature [18], we can compose the two traits, by *adding glue-code* to implement methods  $x$ ,  $y$  and  $of$ . This approach is verbose but very powerful, as illustrated by Wang et al. [40].

```
1 //idealized state of the art trait language, not 42
2 class PointAlgebra=Use pointSum,pointMul, {//not 42 code
3   int x int y//unsatisfactory state of the art solution
4   constructor PointAlgebra(int x, int y){ this.x=x this.y=y }
5   method int x(){return x;}//repetitive code
6   method int y(){return y;}// in traits terminology, this is all "glue code"
7   static method This of(int x, int y){return new PointAlgebra(x,y);}
8 }
```

With a slightly different syntax, this approach is available in both Scala and Rust, and they both require glue code. It has some advantages, but also disadvantages:

- **Advantages:** This approach is associative and commutative, even self instantiation can be allowed if the trait requires a static method returning **This**. The class will then implement the methods returning **This** by forwarding a call to the constructor.
- **Disadvantages:** The class needs to handle all the state, even state conceptually private to a trait. Moreover, writing such obvious code to close the state/fixpoint in the class with the constructors and fields and getter/setters and factories is tedious and error prone; such code could be automatically generated [40].

### 4.2 Our proposed approach to State: Coherent Classes

In  $42_\mu$  there is no need to write down constructors and fields. In fact, in  $42_\mu$  there is not even syntax for those constructs! The intuition is that a class where *all* abstract methods can be seen as field getters, setters, or factories, is a *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 are not coherent. That is, the abstract methods cannot be automatically recognised as factory, getters or setters. Methods recognised as factory, getters and setters are called *abstract state operations*.

A definition of coherent classes is given next, and is formally modelled in Appendix A:

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

Note how the single, abstract static method acts as a *factory method*. The signature of the factory method plays an important role, since abstract state operations are identified by using the names of the factory method arguments. The idea of creating objects in a single atomic step by providing a value for all their fields is well explored (such as with primary constructors in Scala) and does not limit the freedom of programmers to specify personalised initialisation strategies. A static method can freely compute concrete field values before creating objects. Appendix B.4 discusses usability implications of this pattern.

While getters and setters are fundamental operations, it is possible to support more operations. For example:

- **method This** withX(int that) may create a new instance that is like **this** except that field x now has value that. Those kinds of methods performs functional field updates and are called *withers*.
- **method This** clone() may do a shallow clone of the object.

The concept of ‘abstract state operations’ is novel, and we think it is a promising area for further research. The work by Wang et al. [40] explores a particular set of such abstract state operations, but we suspect there are more unexplored possible options that could be even more beneficial.

**Points in  $42_\mu$ :** In  $42_\mu$  and with our approach to handle the state, `pointSum` and `pointMul` can indeed be directly composed. This works because the resulting class is coherent.

```
1 PointAlgebra= Use pointSum,pointMul //no glue code needed
```

**Improved solution:** So far the current solution still repeats 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 declaration by abstracting the common declaration into a trait `p`:

```
1 p= { method int x() method int y()
2   static method This of(int x,int y)
3   }
4 pointSum= Use p, {
5   method This sum(This that){
6     return This.of(this.x()+that.x(),this.y()+that.y());
7   }}
8 pointMul= Use p, {
9   method This mul(This that){
10    return This.of(this.x()*that.x(),this.y()*that.y());
11  }}
12 pointDiv= ...
13 PointAlgebra= Use pointSum,pointMul,pointDiv,...
```

Now the code is fully modularized, that is: each trait defines exactly one method and contains its abstract dependencies. In this way it can be modularly composed with any code requiring such a method.

**Case Study 1:** In order to evaluate our approach we performed a case study: we consider 4 different operations `Sum`, `Subtraction`, `Multiplication` and `Division`. These operations can be combined in 16 different ways. We wrote this example in four different styles: (a) Java7 (115 lines), (b) Classless Java (82 lines), (c) Scala (81 lines) and (d)  $42_\mu$  (32 lines).<sup>4</sup> We chose Classless Java [40] since it is a novel approach allowing Java8 default interface methods to encode traits in Java. We then chose Java7, that lacks the features needed to encode traits, to show the impact of this feature. Finally, the comparison with Scala is interesting since it has good support for

<sup>4</sup> Since we want to focus on the actual code, while counting line numbers we *omit* empty lines and lines containing only open/closed parenthesis/braces.

## Separating Use and Reuse to Improve Both

traits, and using abstract types, it is possible to support the **'This'** type. Rust is similar to Scala in this regard; we believe we would get similar results by comparing against either Scala or Rust.

Language	Lines of code	N. of members	N. classes/traits
Java7	$115 = 6 + 5 * 4 + 7 * 6 + 9 * 4 + 11$	50	16
Classless Java	$82 = 3 + 3 * 4 + 5 * 6 + 7 * 4 + 9$	34	16
Scala	$81 = 5 + 3 * 4 + 4 * 16$	40	$21 = 16 + 4 + 1$
$42_{\mu}$	$32 = 4 + 3 * 4 + 1 * 16$	7	$21 = 16 + 4 + 1$

We observed that in Java7 we had to duplicate<sup>5</sup> 28 method bodies across the 16 classes. Of these, 11 method bodies were duplicated because Java does not support multiple inheritance and the remaining 17 bodies had to be duplicated to ensure that the right type is returned by the method. Those could be avoided if Java supported the **'This'** type. On the other hand, the solution in  $42_{\mu}$  was much more compact since we could efficiently reuse traits (this is why the number of top-level concepts in  $42_{\mu}$  was larger i.e. 21 due to the presence of traits in this solution). In detail, Java required 6 lines for the initial `Point` class, 5 lines for each of the 4 arithmetic operations, 7 lines for each of the 6 combinations of two different operations, 9 lines for each of the 4 combinations of three different operations and finally 11 lines for the class with all four operations. The solution in Classless Java was slightly smaller than Java7, but was still longer than the  $42_{\mu}$  solution: it still had to redefine the `sum`, `sub` and other operations in each of the classes. Here the limited support for the **'This'** type is to blame, thus Classless Java also has 28 duplicated method bodies.

Finally, we compare it with a Scala solution. There is no need for duplicate method bodies in Scala. However, for **'This'** instantiation we need to define abstract methods, that will be implemented in the concrete classes. The Scala solution has the same exact advantages of our proposed solution, and the declaration of the trait is about the same size: 5 (point state) + 3 \* 4 (point operations). However the glue code (the code needed to compose the traits into usable classes) is quite costly: 4 lines for each of the 16 cases. In  $42_{\mu}$  a single line for each case is sufficient.

This example is the best-case scenario for  $42_{\mu}$ : where a maximum level of reuse is required since we considered the case where all the 16 permutations needed to be materialized in the code. In all our case studies, to make a meaningful comparison, we formatted all code in a readable and consistent manner; on the other hand for space limitations, the code snippets presented in the article are formatted for compactness.

### 4.3 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:

```
1 colored= { method Color color() }  
2 CPoint= Use pointSum,colored //Fails: class not coherent
```

<sup>5</sup> A duplicate body is repetition of identical code (may have different types in its scope/environment). The first occurrence is not counted.

This first attempt does not work: the abstract color method is not a getter for any of the parameters of `static method This of(int x,int y)`.

A solution is to provide a richer factory:

```

1 CPoint= Use pointSum,colored,{
2   static method This of(int x,int y){return This.of(x,y,Color.of(/*red*/));}
3   static method This of(int x,int y,Color color) }

```

where we assume support for overloading based on different numbers 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, we can use `this.withX(newX).withY(newY)`, instead of writing `This.of(...)`, in order to preserve the color from `this`. This solution is better but still not satisfactory since the color from that is ignored.

**A better design:** We can design trait `p` for reuse and extensibility by adding an abstract `merge(This)` method as an extensibility hook; `colored` can now define color merging. Using withers we can merge colors, or any other kind of state following this pattern.

```

1 p= { method int x() method int y() //getters
2   method This withX(int that) method This withY(int that)//withers
3   static method This of(int x,int y)
4   method This merge(This that) //new method merge!
5   }
6 pointSum= Use p, {
7   method This sum(This that){
8     return this.merge(that).withX(this.x()+that.x()).withY(this.y()+that.y());
9   }
10 colored= {method Color color()
11   method This withColor(Color that)
12   method This merge(This that){ //how to merge color handled here
13     return this.withColor(this.color().mix(that.color()));
14   }}
15 CPoint= /*as before*/

```

**Independent Extensibility:** Of course, quite frequently there can be multiple independent extensions [41] that need to be composed. Lets suppose that we could have a notion of `flavored` as well. In order to compose `colored` with `flavored` we would need to compose their respective merge operations. To this aim `Use` is not sufficient. To combine the implementation of two different implementation of methods, we introduce an operator called `super`, that makes a method abstract and moves the implementation to another name. This is very useful to implement super calls and to compose conflicting implementations. Consider the simple `flavored` trait:

```

1 flavored= {
2   method Flavor flavor() //very similar to colored
3   method This withFlavor(Flavor that)
4   method This merge(This that){ //merging flavors handled here
5     return this.withFlavor(that.flavor());} //inherits "that" flavor

```

In order to merge `colored` and `flavored` we use `super` to introduce method selectors `_1merge` and `_2merge` to refer to the version of `merge` as defined in the first/second element of `Use`.

## Separating Use and Reuse to Improve Both

```
1 FCPoint= Use
2   colored[super merge as _1merge], //this leaves merge as an abstract method, and
3   flavored[super merge as _2merge], //copies the bodies into _1merge and _2merge
4   pointSum,{
5     static method This of(int x,int y){
6       return This.of(x,y,Color.of(/*red*/),Flavor.none());}
7     static method This of(int x, int y,Color color,Flavor flavor)
8     method This merge(This that){//merge conflict is solved
9       return this._1merge(that)._2merge(that);} //by calling the two versions
10  }
```

Note how we are relying on the fact that the code literal does not need to be complete, thus we can just call `_1merge` and `_2merge` without declaring their abstract signature explicitly.

In this last example, when we tried to obtain state extensibility, we refactored the code to introduce the `merge(This)` method. This suggests that we had to anticipate the need for state extensibility in order to design our original code. As illustrated by the following example, we can instead rely on the `super` operator to inject the `merge(This)` method when needed.

```
1 p=/*as originally designed: no merge*/
2 pointSum=/*as originally designed: no merge*/
3 merge={method This merge(This that)}
4 pointSumMerge=Use merge, pointSum[super sum as _1sum], {
5   method This sum(This that){return this.merge(that)._1sum(that);}}
6 colored=/*as before, with merge implementation*/
7 CPoint= /*as before, but using pointSumMerge*/
```

**Case Study 2:** To understand how easy it is to extend the state in this way we compare the former code with an equivalent version in Java. For this example, in Java we encode `Point` with the fields but no operations, `PointSum` reuses `Point` adding a functional `sum` operation, `CPoint` reuses `PointSum` with a `Color` field and `FCPoint` reuses `CPoint` with a `Flavour` field. This second case study represents a *worst case scenario* for  $42_{\mu}$  against Java because we model just a single chain of reuse, easily supported in plain Java by single inheritance. Like the previous experiment, we still found that the Java solution was longer (47 lines) than that in  $42_{\mu}$  (33 lines). This is caused by the absence of support for the `'This'` type, where the withers in each of the `CPoint/FCpoint` classes had to be repeated to make sure that the returned type will be correct (the number of members in Java were 27 while 24 (3 less) in  $42_{\mu}$ ).

Complex patterns in Java<sup>6</sup> allow supporting the `'This'` type and `'This'` type instantiation but they require a lot of set-up code. We experimented with those patterns, but it soon became very clear that the resulting code of this approach would have been even larger; albeit without duplicated code. Note how the Java code is less modular than the  $42_{\mu}$  code, since `Colored` and `Flavored` do not exist as individual concepts.

<sup>6</sup> Combining the ones used in those works [34, 38], with abstract methods to allow self instantiation as in [41].

We also compare with a solution in Scala, offering the same level of reuse and code modularity of the  $42_\mu$  solution, but again it is more verbose and requires more members (31): an indication that it may be logically heavier too. We define the main `tPoint` trait (8 lines), the `tPointSum` operation (3), the two `tColored` and `tFlavored` traits ( $6 * 2$ ) and the `CPoint` and `CFPoint` classes ( $12 + 18$ ). The major benefit of  $42_\mu$  is the reduction of the amount of glue-code needed to generate `CPoint` and `CFPoint` ( $4 + 9$ ).

The results for the second experiment are presented below.

Language	Lines of code	N. of members	N. of classes or traits
Java	$47 = 10 + 9 + 13 + 15$	27	6
Scala	$53 = 8 + 3 + 6 * 2 + 12 + 18$	31	6
$42_\mu$	$33 = 7 + 3 + 5 * 2 + 4 + 9$	24	6

## 5 Family Polymorphism by Disconnecting Use and Reuse

A nested class is just another kind of member in a code literal. In Java and Scala if a subclass declares a nested class with the same name of a nested class of a superclass, the parent declaration is simply hidden. The main idea of family polymorphism (FP) [12, 16, 21, 24, 25, 32, 36] is to instead consider such definition a form of *overriding*, called *further extension*. That is, the following Java code is ill typed:

```

1 abstract class A{static class B{..} abstract B m();}
2 class AA extends A{static class B{..} B m(){..}}//Error: Invalid overriding
3 //method AA.m() return type is AA.B, that is unrelated to A.B

```

In the FP approach, class `AA.B` would further extend `A.B`, thus the overriding of method `A.m` would be accepted. We extend  $42_\mu$  with nested classes, so that by composing code with **Use**, nested classes with the same name are recursively composed. The corresponding code in  $42_\mu$  would work, and behave like further extension in FP.

```

1 a={B={..} method B m()}
2 AA= Use a, {B={..} method B m(){..}}

```

For simplicity, we discuss nested classes but not nested traits: and all traits and code composition expressions are still at top level. In this way all dependencies are about top level names, allowing the type system to consider the class table as a simple map from (nested) type names (such as `A` and `A.B.C`) to their definition.

There are a lots of different forms of rename in literature [1, 10, 16]. Here we introduce a simple variant to rename nested types to other nested types. For example:

```

1 t={ method B m() B={ method B mb() } }
2 D= t[rename B into C]

```

would flatten to:

```

1 t={ method B m() B={ method B mb() } }
2 D={ method C m() C={ method C mb() } }

```

The rename only influences its argument. Since traits do not induce nominal types, we can consistently change their internally used names without breaking any code. The full  $L_{42}$  offers many other kinds of renames, but we do not need them to show our next example.

## Separating Use and Reuse to Improve Both

### Application to the expression problem. Case Study 3:

The above extensions lets us challenge the expression problem [39], with the requirements exposed in [41]. In the expression problem we have data-variants and operations and we can *extend our solution in both dimensions*, by adding new data-variants and operations. We aim to *combine independently developed extensions* so that they can be used jointly. To be modular, extensions will preserve *type safety* and allow *separate compilation* (no re-type-checking), while avoiding *duplication of source code*.

Following closely the example of Zenger and Odersky [41], we consider a language where the expressions `Exp` can be `Num` (for number literal), `Plus` (for binary plus operator) and `Neg` (for unary minus). We then proceed to define operations `show` to convert them into strings, `eval` to compute their numeric values and `double` to double their containing `Nums`. We thus have 3 classes, 1 interface, the definition of the state, and 3 operations. We model this as a table of features, as in [16]: a (3 classes + 1 interface)\*(1 state + 3 operations) table composed by 16 traits. The features are atomic: they exactly declare the state of a class or define a single operation for a single class.  $42_{\mu}$  avoids the large amount of abstract declarations that clutters the solution in [16]. Intuitively, we would like our traits to look like the following:

```
1 evalPlus= Use plus, { //eval operation for Plus data-variant
2   Exp= {interface
3     method int eval()}
4   Plus= {implements Exp
5     method int eval(){
6       return this.left().eval()+this.right().eval();}}
```

`evalPlus` uses the trait `plus` to import the state (the `left()` and `right()` methods) and defines the `eval()` method from interface `Exp`. But, if we were to declare those explicitly, we would repeat `Exp`, the abstract declaration of `eval()` and '`implements Exp`' for all data-variants. To avoid this duplication, we write the trait `eval` with a placeholder `T` nested class, that can then be renamed into the corresponding data-variant. Thus, our source code is as follows; First we declare the 4 traits to represent the state:

```
1 exp= {   Exp= {interface}   T= {implements Exp}}

1 num= Use exp[rename T into Num],{ //T is renamed to Num and summed with
2   Num= {method int value() static method Num of(int value)}} // this Num

1 plus= Use exp[rename T into Plus], {
2   Plus= {method Exp left() method Exp right()
3     static method Plus of(Exp left,Exp right)}}

1 neg= Use exp[rename T into Neg],{
2   Neg= {method Exp term() static method Neg of(Exp term)}}}
```

Here we define a trait for each data-variant. Each trait will contain its version of `Exp` and a specific kind of expression, with its state. Next, we define the operation `eval` for all the data-variants. The former solutions in [16] required repeating the state declaration of the data-variant in each operation, while we can just import it.

```
1 eval= {Exp= {interface   method int eval()}   T= {implements Exp}}

1 evalNum= Use num, eval[rename T into Num],{ //just the implementation
2   Num= { method int eval(){return this.value();} }}//of the specific method
```



```

1 evalPlus= Use plus, eval[rename T into Plus], {
2   Plus= { method int eval(){ return this.left().eval()+this.right().eval();} }}
1 evalNeg= Use neg, eval[rename T into Neg], {Neg={ method int eval(){..}}}
    
```

The `show` operation can be trivially defined following exactly the same pattern (omitted here for space reasons). The operation `double` is a challenge for some proposed solutions to the expression problem, as explained by Zhang and Oliveira [42]. The `double` operation is called a *transformation*: an operation from  $\text{Exp}$  to  $\text{Exp}$ . Thanks to  $42_\mu$ 's separation between use and reuse, together with support for self-instantiation of nested classes `double` does not need any special attention and can be written just like `eval` and `show`.

```

1 double= {      Exp= {interface      method Exp double()}      T= {implements Exp} }
1 doubleNum= Use num, double[rename T in Num],{
2   Num= { method Exp double(){return Num.of(this.value()*2);} }}
1 doublePlus= Use plus,double[rename T in Plus],{
2   Plus= { method Exp double(){
3     return Plus.of(this.left().double(),this.right().double());} }}
1 doubleNeg=....
    
```

Here we define a trait for each data-variant implementing the operation `double()`. Again, each trait will contain its version of `Exp` with `double()` and a specific kind of expression, with the implementation for `double()` for that specific kind.

Our third case study compares with the results presented in Scala [41]. The proposed solution is not fully modularized as a table, so in order to make a more close comparison, we provide an alternative version where we isolate all the units of behaviour as is done in  $42_\mu$ .

	lines	methods	
Original Scala	52	15 = 12 + 3	Scala uses 12 = 4 × 3 methods plus 3 extra factory methods (for <code>double</code> ). We use 12 = 4 × 3 methods plus our abstract state: 4 getters and 3 factories.
Scala isolated units	78	15 = 12 + 3	
Scala glue-code	27	3	
42 traits	48	19 = 4 × 3 + 7	
42 classes	3	0	

As we can see, encoding atomic units in Scala is more verbose, but more importantly, in  $42_\mu$  we can just define a class supporting any subset of operations and data-variants by listing the desired traits: for example, a solution for `Num` and `Plus` (but not `Neg`) with `eval` and `double` would look like this: `Example= Use evalNum,evalPlus,doubleNum,doublePlus`. The composition of all our traits would just requiring listing all of the relevant behaviour; reasonably formatted, it could take up to 3 lines. On the other hand, the presented solution in Scala requires 27 lines of glue code to put the traits together. This means that a full Scala solution *requiring a single instantiation with all the traits* would be 78+27 = 105 lines. If we were to require more instantiations with a different subset of traits, the glue code would dominate the line count, and the Scala solution would end up being up to 9 times heavier than the  $42_\mu$  one (if all 64 permutations were required).

## Separating Use and Reuse to Improve Both

The line count for  $42_\mu$  is very predictable: after defining `exp` (3) and the state traits (4 + 6 + 5) for each of the three operations (`eval`, `show`, `double`) we just needed 4 lines to declare the operation in the interface, and 2 lines for each of the 3 data-variants.

Following [4I], after `double` we present an implementation of `equals`. Their solution involved double dispatch to avoid casting. To show understandable code, we show a simpler solution with a guarded cast (sometime called a typecase).<sup>7</sup> The idea is that since every data-variant contains the same "cast" logic, we can modularize it into an `equals` trait; `equals` in [4I] is complex and requires glue code.

```
1 equals= {
2   Exp= {interface      method Bool equals(Exp that)}
3   T= {implements Exp
4     method Bool exactEquals(T that)
5     method equals(that){
6       if(T instanceof This){return this.exactEquals(that);}else{ return false;}}
```

```
1 equalsNum= Use num, equals[rename T into Num],{
2   Num= {method Bool exactEquals(Num that){
3     return this.value().equals(that.value());} }}
```

```
1 equalsPlus= Use plus, equals[rename T into Plus],{
2   Plus= {method Bool exactEquals(Plus that){
3     return this.left().equals(that.left()) && this.right().equals(that.right());
4   } }}
```

```
1 equalsNeg= Use neg, equals[rename T into Neg],{
2   Neg= {method Bool exactEquals(Neg that){
3     return this.term().equals(that.term());} }}
```

	lines	methods	The Scala code here can be made fully isolated with little extra syntactic cost. The original Scala eq is 40 lines and contains a part of the glue code mixed inside. The isolated version is 31 lines and to merge all the operations together in Scala, it takes 29 lines of glue code. Note that this is mostly the same glue code from before (27 lines), that needs to be manually adapted.
Original Scala equals	40	10	
Isolated Scala equals	31	10	
Scala equals instance	29	3	
42 trait eq d-dispatch	21	6	
42 class dd instance	22	11	
42 traits eq Cast	13	6	
42 class cast instance	3	0	

In  $42_\mu$  we are more compact than Scala both when using the double dispatch (21 + 22 vs. 31 + 29) or the guarded cast (13 + 3 vs. 31 + 29). To instantiate the double dispatch version in  $42_\mu$  we need 22 lines of glue code. We could remove such glue code using features from the full 42 language, but here we stick to only the features presented in this paper. The interesting point is that the nature of our needed glue code is different with respect to the Scala glue code: Scala requires lots of trait multiple inheritance declarations to explicitly merge nested traits with the same name, while in  $42_\mu$  we mostly need to add the negative cases for the double dispatch (such as `Sum={method Bool equalToNum(Num that){return false;}}`).

<sup>7</sup> The interested reader can find a  $42_\mu$  implementation of `equals` with double dispatch in the appendix.

## 6 Summary of formalisation

In Appendix A we formalise  $42_\mu$ ; in addition to conventional soundness, we discuss detailed behaviour and soundness of the compilation process itself; a similar property was called meta-level-soundness in [37]. This property ensures that flattening strictly reduces the number of type errors. In turn, this ensure that reusing a trait cannot induce new type errors. This property was already proved in [37]; here the proof is smoother thanks to our simpler formalisation. Our process requires traits to be well-typed before being reused, however code literals are not required to be well typed before flattening. This design supports mutually recursive types without having to predict the structure of the flattened code, as was needed in [16].

## 7 Related Work

Literature on code reuse is too vast to let us do justice to it in a few pages. In particular, we were unable to discuss all the variations on family polymorphism. Our work is inspired by traits [18], which in turn are inspired by module composition languages [1].

### 7.1 Separating Inheritance and Subtyping

In languages like Cecil [14] and PolyToil [13], classes are not types: it is a more radical solution to ‘inheritance implies subtyping’, and equivalent to a restricted version of  $42_\mu$  where only interface names can be used as types. This complicates typing of **this**, and may prevent any useful application of the **This** type (PolyToil uses polymorphism to support it). Those approaches would ban the following code, since A is not a type:

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

Cecil syntactic sugar counters this issue; advantages and pitfalls are discussed in Appendix B (especially B.2).

$42_\mu$  is directly inspired by the 3 independently designed research languages as already mentioned: TraitRecordJ (TR)[6], Package Templates (PT)[26] and DeepFJig(DJ) [16]. We synthesize the best ideas of those very different designs, while at the same time coming up with a simpler and improved design for separating subclassing from subtyping, which also addresses various limitations of those 3 particular language designs. TR, DJ, and PT are research projects, aiming to be platforms to experiment concepts, not to expose a compact syntax to programmers; instead of using case studies to compare  $42_\mu$  against TR, DJ, and PT, in the following we compare various aspects of the language designs on a more theoretical level. We **bold** 3 properties where one approach shines the most, and we squiggle 3 properties where one approach is more lacking.

- **A simple uniform syntax for code literals.** Between those tree approaches, **DJ** is best in this sense: TR has separate syntax for class literals, trait literals and record literals. PT is built on top of Java, thus, it must support many different syntactic forms.  $42_\mu$  relies on DJ’s approach but, thanks to our novel representation of state,  $42_\mu$  offers a much simpler and uniform syntax than DJ, TR, and PT.

## Separating Use and Reuse to Improve Both

- **Reusable code cannot be “used” (that is be instantiated or used as a type).** This happens in TR and in PT, but not in DJ. In DJ, to allow reusable code to be directly usable, classes introduce nominal types in an unnatural way: the type of `this` is only `This` (sometimes called `<>`) and not the nominal type of its class. That is in DJ `A={method A m(){return this;}}` is not well typed. This is because `B= Use A` flattens to `B={method A m(){return this;}}`, which is clearly not well typed. Looking at this examples makes it clear why we need reusable code to be agnostic of its name as in TR, PT, and  $42_\mu$ : either reusable code does not correspond to a type name (as in TR, PT, and  $42_\mu$ ) or all code is reusable and usable, and all code needs to be awkwardly agnostic of its name, as in DJ.
- **Requiring abstract signatures is a left over of module composition mindset.** TR and DJ comes from a tradition of functional module composition, where modules are typed in isolation under an environment, and then the composition is performed. As we show in this work, this ends up requiring verbose repetition of abstract signatures which (for highly modularized code) may end up constituting most of the program. Java (and thus PT, as a Java extension) show us a better way: names are understood from their reuse context. The typing of PT offers the same advantages of the  $42_\mu$  typing model, but is more indirect. This may be caused by the heavy task of integrating with full Java. Recent work based on TR is trying to address this issue too [17].
- **Composition algebra.** The idea of using composition operators over atomic values as in an arithmetic expression is very powerful, and makes it easy to extend languages with more operators.  $42_\mu$ , DJ, and TR embrace this idea, while PT takes the traditional Java/C++ approach of using an enhanced class/package declaration syntax. The typing strategy of PT also seems to be connected with this decision, so it would be hard to move their approach to a composition algebra setting.
- **Complete ontological separation between use and reuse.** While  $42_\mu$ , TR, DJ, and PT all allow separating inheritance and subtyping only  $42_\mu$  and TR properly enforce separation between use (classes and interfaces) and reuse (traits). In DJ all classes are both units of use and reuse (however, subtyping is not induced). PT imports all the complexity of Java: it is possible to separate use and reuse, the model has powerful but non-obvious implications where Java `extends` and PT are used together.
- **Naming the self type, even if there is none yet.** TR is lacking here, while  $42_\mu$ , DJ and PT both allow a class to refer to its name; albeit this is less obvious in PT since both a package and a class have to be introduced to express it. This allows encoding binary methods, expressing patterns like withers or fluent setters and to instantiate instances of (future) classes using the reused code.

## 7.2 Implications for Family Polymorphism

Our `Use` operator is similar to deep mixin composition [19, 22, 41] and family polymorphism [21, 24, 25, 36], but is symmetric while the operator `super` offers flexible explicit conflict resolution. Our presented solution to the expression problem improves

over existing solutions in the literature, where one close contender is DJ [16]: our gain over their model is based on our relaxation over abstract signatures. A similar syntax can be achieved with the Scandinavian style [20], or with the work of Nystrom (Jx [31] and J& [32]), where the composition behaves similarly to our sum operator. Both Jx, J&, and the virtual classes of Ernst [21] make use of dependent types. As in .FJ and ^FJ [24, 25, 34, 36], we do not need sophisticated types. The work on DJ [16] contains an in-depth comparison between various FP approaches, including an example written in .FJ syntax synthesizing the difficulty of supporting FP while keeping Use and Reuse connected:

```
1 class A{static class B{int f1;} int k(.B x){ return x.f1;}}
2 class AA extends A{static class B{int f2;} int k(.B x){return x.f2+new .B().f2;}}
```

The syntax `.B` denotes a relative path, that is, the class `B` in scope. In FP `AA.B` further extends `A.B`: it is implicitly considered a subclass of `A.B`, adding the field `f2`. Consider now the following code:

```
1 new AA().k(new AA.B())//well-typed
2 new A().k(new A.B())//well-typed
3 A a=new AA(); //well-typed assuming AA is a subtype of A
4 a.k(new A.B())//runtime error: A.B.f2 does not exist
```

In the sound .FJ type system the last method invocation `illtyped` even though `AA.B` is a subtype of `A.B`. With minor changes, others [12, 21, 25, 32] support this example in the same way. Inheritance implying subtyping is broken only in a controlled way, and it is allowed whenever it does not lead to unsoundness. Recent work on **ThisType** [33, 35] also continues in this line. In those works, “subtyping by subclassing” is preserved: those designs aim to retain the programming model of mainstream OOP languages and backwards compatibility.  $\lambda_2$  is instead a radical departure from mainstream OOP, hoping to improve the mechanisms for use and reuse in OOP and unlock new ways to design software.

From a different perspective, we can say that traditional implementations of family polymorphism are still heavily influenced by the “inheritance implies subtyping” model. We believe that this is a major source of complexity in the type systems of those approaches: they need to track calls, and enforce that the *family* of the receiver and the argument is the same. Because we separate inheritance from subtyping we liberate ourselves from tricky issues that arise in such type systems, and can provide a simpler model of family polymorphism, soundly supported by a straightforward nominal type system: by disconnecting use and reuse we outlaw `A a=new AA()`. In  $\lambda_{2\mu}$  this also reduces the expressive power a little, but in the full  $\lambda_2$  language, as well as in DJ, the operator *redirect* allows us to write code that is parametric on families of data types. To the same aim, .FJ relies on generics.

Support for FP strictly includes support for the ‘**This**’ type and self instantiation. Scala allows *encoding* further extension/deep mixin composition, but it requires doing it explicitly, growing the amount of required glue-code.

## Separating Use and Reuse to Improve Both

### 7.3 State and traits

The original trait model [18] has no self instantiation and avoids any connection between state and traits. Since it was applied to a dynamic language, the relation with the `This` type is unclear.

The idea of abstract state operations emerged from Classless Java [40]. This approach offers a clean solution to handle state in a trait composition setting. Note how abstract state operations are different from just hiding fields under getters and setters: in our model the programmer simply never has to declare what information is stored in fields. The state is computed by the system as an overall result of the whole code composition process.

In the literature there have been many attempts to add state in traits and in module composition languages:

- The simplest solutions have no constructors, and fields start with **null** (or zero/false). In this setting fields are another kind of (abstract) member, and two fields with identical types 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)`. This approach is commutative and associative. However, objects are created "broken" and the user is trusted with fixing them. While it is easy to add fields, the load of initializing them is on the user; moreover all the objects are intrinsically mutable, preventing a functional programming style.
- **Constructors compose fields:** Here a canonical constructor (as in FJ) taking a parameter for each field and just initializing the fields is assumed to be present. It is easy to add fields, however this model (used by [28]) is associative but not commutative: composition order influences field order, and thus the constructor signature. Self instantiation is also not possible since the signature of the constructors change during composition.
- **Constructors can be composed if they offer the same exact parameters:** In this model, traits declare fields and constructors initialize their fields using any kind of 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 in DJ to allow self instantiation. It is associative and mostly commutative: composition order only influences execution order of side effects during construction. However trait composition requires identical constructor signatures: this hampers reuse, and if a field is added, its initial value needs to be synthesized from the other parameters.

### 7.4 Tabular comparison of many approaches

In this table we show if some constructs support certain features: direct instantiation (as in `new C()`), self instantiation (as in `new This()`), is this construct a 'unit of use?', a 'unit of reuse?', does using this construct introduce a type? and is the induced type the type of `this`?, support for binary methods, does inheritance of this construct induce subtyping?, is the code of this construct required to be well-typed before being inherited /imported to a new context? is it required to be well-typed before being

composed with other code? Y and N means yes and no; we use “-” where the question is not applicable to the current approach. For example the original trait model was untyped, so typing questions make no sense there.

	direct instantiation	self instantiation	unit of use	unit of reuse	introduces a type	induced type is the type of <b>this</b>	binary methods	inheritance induces subtyping	typed before imported	typed before composed
Java/Scala classes	Y	N	Y	Y	Y	Y	N	Y	Y	N
Java8 interfaces	N	N	N	Y	Y	Y	N	Y	Y	N
Scala traits	N	N	N	Y	Y	Y	-	Y	Y	N
Original traits	N	N	N	Y	-	-	-	N	-	-
TR	N	N	N	Y	N	-	N	N	Y	Y
42 <sub>μ</sub> traits	X	Y	X	Y	N	-	Y	N	Y	N
42 <sub>μ</sub> classes	Y	Y	Y	N	Y	Y	Y	-	Y	-
Module composition	-	-	Y	Y	-	-	-	-	Y	Y
DJ classes	Y	Y	Y	Y	Y	X	Y	N	Y	Y
PT	N	Y	N	Y	N	-	-	N	Y	N

## 8 Conclusions, extensions and practical applications

In this paper we explained a simple model to radically decouple inheritance/code reuse and subtyping. Our decoupling does not make the language more complex: we replace the concept of abstract classes with the concept of traits, while keeping the concepts of interfaces and final classes. Concrete non final classes are simply not needed in our model. Thus, we believe that 42<sub>μ</sub> is beneficial for code reuse in important cases without having negative impacts on the general programming experience. The interested reader can find a detailed discussion about this in Appendix B.

The model presented here is easy to extend. More composition operators can be added in addition to **Use**. Variants of the sophisticated operators of DJ are included in the full 42 language. Indeed we can add any operator respecting the following:

- When the operator fails it needs to provide an error that will be reported to the programmer.
- When only well typed code is taken in input, if a result is produced, such result is also well typed.
- When the result is not well typed, the type error must be traced back to a fault in the input.

Our simplified model represents the conceptual core of 42: a novel full blown programming language. In full 42 code literals are first class values, thus we do not need explicit names for traits: they are encoded as methods returning a code literal.

## Separating Use and Reuse to Improve Both

### References

- [1] Davide Ancona and Elena Zucca. “A calculus of module systems”. In: *Journal of functional programming* 12.02 (2002), pages 91–132.
- [2] Eyvind W. Axelsen and Stein Krogdahl. “Package templates: a definition by semantics-preserving source-to-source transformations to efficient Java code”. In: *GPCE’12*. 2012.
- [3] Eyvind W. Axelsen, Fredrik Sørensen, Stein Krogdahl, and Birger Møller-Pedersen. “Challenges in the Design of the Package Template Mechanism”. In: *Trans. Aspect-Oriented Software Development* 9 (2012), pages 268–305.
- [4] Alan Bawden et al. “Quasiquotation in Lisp.” In: *PEPM’99*. 1999.
- [5] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. “Stateful Traits”. In: *Advances in Smalltalk: 14th International Smalltalk Conference, ISC 2006, Prague, Czech Republic, September 4-8, 2006, Revised Selected Papers*. Edited by Wolfgang De Meuter. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 66–90. ISBN: 978-3-540-71836-9. DOI: 10.1007/978-3-540-71836-9\_4.
- [6] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. “Implementing Software Product Lines Using Traits”. In: *SAC ’10*. 2010.
- [7] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. “Implementing type-safe software product lines using parametric traits”. In: *Science of Computer Programming* 97, Part 3 (2015), pages 282–308.
- [8] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocchio. “TraitRecordJ: A programming language with traits and records”. In: *Science of Computer Programming* 78.5 (2013), pages 521–541.
- [9] Joshua Bloch. *Effective Java (2Nd Edition) (The Java Series)*. 2nd edition. Prentice Hall PTR, 2008.
- [10] Gilad Bracha. “The programming language jigsaw: mixins, modularity and multiple inheritance”. PhD thesis. Dept. of Computer Science, University of Utah, 1992.
- [11] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. “On Binary Methods”. In: *Theory and Practice of Object Systems* 1.3 (1996).
- [12] Kim B Bruce, Martin Odersky, and Philip Wadler. “A statically safe alternative to virtual types”. In: *European Conference on Object-Oriented Programming*. Springer. 1998, pages 523–549.
- [13] Kim B Bruce, Angela Schuett, and Robert Van Gent. “PolyTOIL: A type-safe polymorphic object-oriented language”. In: *European Conference on Object-Oriented Programming*. Springer. 1995, pages 27–51.
- [14] Craig Chambers and Gary T Leavens. “Typechecking and modules for multimethods”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 17.6 (1995), pages 805–843.



- [15] William R. Cook, Walter Hill, and Peter S. Canning. “Inheritance is Not Subtyping”. In: *POPL '90*. 1990.
- [16] Andrea Corradi, Marco Servetto, and Elena Zucca. “DeepFJig - Modular composition of nested classes”. In: *Journal of Object Technology* 11.2 (2012), 1: 1–42. DOI: 10.5381/jot.2012.11.2.a1. URL: <http://dx.doi.org/10.5381/jot.2012.11.2.a1>.
- [17] Ferruccio Damiani, Reiner Hähnle, Eduard Kamburjan, and Michael Lienhardt. “A Unified and Formal Programming Model for Deltas and Traits”. In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2017, pages 424–441.
- [18] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P Black. “Traits: A mechanism for fine-grained reuse”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28.2 (2006), pages 331–388.
- [19] Erik Ernst. “Propagating class and method combination”. In: *ECOOP*. Volume 99. 6. Springer. 1999, pages 67–91.
- [20] Erik Ernst. “The expression problem, Scandinavian style”. In: *ON MECHANISMS FOR SPECIALIZATION* (2004), page 27.
- [21] Erik Ernst, Klaus Ostermann, and William R Cook. *A virtual class calculus*. Volume 41. 1. ACM, 2006.
- [22] DeLesley Hutchins. “Eliminating distinctions of class: Using prototypes to model virtual classes”. In: *ACM SIGPLAN Notices*. Volume 41. 10. ACM. 2006, pages 1–20.
- [23] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. “Featherweight Java: a minimal core calculus for Java and GJ”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pages 396–450.
- [24] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. “Lightweight family polymorphism”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2005, pages 161–177.
- [25] Atsushi Igarashi and Mirko Viroli. “Variant path types for scalable extensibility”. In: *ACM SIGPLAN Notices* 42.10 (2007), pages 113–132.
- [26] Stein Krogdahl, Birger Møller-Pedersen, and Fredrik Sørensen. “Exploring the Use of Package Templates for Flexible Re-use of Collections of Related Classes”. In: *Journal of Object Technology* 8.7 (2009), pages 59–85.
- [27] Giovanni Lagorio, Marco Servetto, and Elena Zucca. “Featherweight Jigsaw: A Minimal Core Calculus for Modular Composition of Classes.” In: *ECOOP*. Springer. 2009, pages 244–268.
- [28] Giovanni Lagorio, Marco Servetto, and Elena Zucca. “Featherweight Jigsaw - Replacing inheritance by composition in Java-like languages”. In: *Inf. Comput.* 214 (2012), pages 86–111. DOI: 10.1016/j.ic.2012.02.004.
- [29] Wilf LaLonde and John Pugh. “Smalltalk: Subclassing Subtyping Is-a”. In: *J. Object Oriented Program.* 3.5 (Jan. 1991), pages 57–62. ISSN: 0896-8438.

## Separating Use and Reuse to Improve Both

- [30] Robert C Martin. “Design principles and design patterns”. In: *Object Mentor* 1.34 (2000).
- [31] Nathaniel Nystrom, Stephen Chong, and Andrew C Myers. “Scalable extensibility via nested inheritance”. In: *ACM SIGPLAN Notices*. Volume 39. 10. ACM. 2004, pages 99–115.
- [32] Nathaniel Nystrom, Xin Qi, and Andrew C Myers. “J&: nested intersection for scalable software composition”. In: *OOPSLA '06*. Volume 41. 10. ACM. 2006, pages 21–36.
- [33] Sukyoung Ryu. “ThisType for Object-Oriented Languages: From Theory to Practice”. In: *ACM Trans. Program. Lang. Syst.* 38.3 (Apr. 2016).
- [34] Chieri Saito and Atsushi Igarashi. “A.: The essence of lightweight family polymorphism”. In: *Journal of Object Technology*. Citeseer. 2008.
- [35] Chieri Saito and Atsushi Igarashi. “Matching ThisType to Subtyping”. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. 2009.
- [36] Chieri Saito, Atsushi Igarashi, and Mirko Viroli. “Lightweight family polymorphism”. In: *Journal of Functional Programming* 18.3 (2008), pages 285–331.
- [37] Marco Servetto and Elena Zucca. “A meta-circular language for active libraries”. In: *Science of Computer Programming* 95 (2014), pages 219–253.
- [38] Mads Torgersen. “The expression problem revisited”. In: *European Conference on Object-Oriented Programming*. Springer. 2004, pages 123–146.
- [39] Philip Wadler. “The expression problem”. In: *Java-genericity mailing list* (1998).
- [40] Yanlin Wang, Haoyuan Zhang, Bruno C d S Oliveira, and Marco Servetto. “Classless Java”. In: *GPCE'16*. 2016.
- [41] Mathhias Zenger and Martin Odersky. “Independently Extensible Solutions to the Expression Problem”. In: *Foundations of Object-Oriented Languages (FOOL)*. Jan. 2005.
- [42] Weixin Zhang and Bruno C d S Oliveira. “EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse”. In: *LIPICs-Leibniz International Proceedings in Informatics*. Volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

## A Formalisation

Here we show a simple formalization for the language we presented so far. We also model nested classes, but in order to avoid uninteresting complexities, we assume that all type names are fully qualified from top level, so the examples shown before should be written like: **This**.Exp, **This**.Sum, etcetera. In a real language, a simple pre-processor may take care of this step.

In most languages, when implementing an interface, the programmer may avoid repeating abstract methods they do not wish to implement, however to simplify our formalization, we consider source code always containing all the methods imported from interfaces. In a real language, a normalisation process may hide this abstraction.<sup>8</sup> We also consider a binary operator sum (+) instead of the nary operator **Use**. Figure 1 contains the complete formalization for  $42_\mu$ : syntax, compilation process, typing, and finally reduction.

### A.1 Syntax

We use  $t$  and  $C$  to represent trait and class identifiers respectively. A trait ( $TD$ ) or a class ( $CD$ ) declaration can use either a code literal  $L$ , or a trait expression  $E$ . Note how in  $E$  you can refer to a trait by name. In full 42, we support various operators including the ones presented before and much more, but here we only show the single sum operator: +. This operation is a generalization to the case of nested classes of the simplest and most elegant trait composition operator [18]. Code literals  $L$  can be marked as interfaces. We use '?' to represent optional terms. Note that the interface keyword is inside curly brackets, so an uppercase name associated with an interface literal is a interface class, while a lowercase one is a interface trait. Then we have a set of implemented interfaces and a set of member declarations, which can be methods or nested classes. The members of a code literal are a set, thus their order is immaterial. If a code literal implements no interfaces, the concrete syntax omits the **implements** keyword.

Method declarations  $MD$  can be instance methods or **static** methods. A static method in  $42_\mu$  is similar to a **static** method in Java, but can be abstract. This is very useful in the context of code composition. To denote a method as abstract, instead of an explicit keyword we just omit the implementation  $e$ .

Finally, expressions  $e$  are just variables, instance method calls or static method calls. Having two different kinds of method calls is an artefact of our simplifications. In the full 42 language, type names are a kind of expression whose type helps to model metaclasses. Our values  $v_{\bar{D}}$  are just calls to abstract static methods: thanks to abstract state, we have no **new** expressions, but just factory calls. Thus values are parametric on the shape of the specific programs  $\bar{D}$ . We then show the evaluation context, the compilation context and full context.

<sup>8</sup> In the full 42 language scoping is indeed supported by an initial de-sugaring, and a normalisation phase takes care of importing methods from interfaces.

## Separating Use and Reuse to Improve Both

$ID ::= t \mid C$	class or trait name
$DE ::= ID = E$	Meta-declaration
$D ::= ID = L$	Declaration
$E ::= L \mid t \mid E+E \mid \dots$	Code Expression
$L ::= \{\mathbf{interface? implements-T} \bar{M}\}$	Code Literal
$T ::= C \mid C.T$	Type
$M ::= \mathbf{static? method} T m(\bar{T}x) e? \mid C = L$	Member
$e ::= x \mid e.m(\bar{e}) \mid T.m(\bar{e})$	Expression
$v_{\bar{D}} ::= T.m(\bar{v}_{\bar{D}})$ , where $m$ is abstract in $\bar{D}(T)$	value
$\mathcal{E}_{\bar{D}} ::= [] \mid \mathcal{E}_{\bar{D}}.m(\bar{e}) \mid v_{\bar{D}}.m(\bar{v}_{\bar{D}}, \mathcal{E}_{\bar{D}}, \bar{e}) \mid T.m(\bar{v}_{\bar{D}}, \mathcal{E}_{\bar{D}}, \bar{e})$	evaluation context
$\mathcal{E}_c ::= [] \mid \mathcal{E}_c + E \mid L + \mathcal{E}_c \mid \dots$	compilation context
$\mathcal{E} ::= [] \mid \mathcal{E} + E \mid E + \mathcal{E} \mid \dots$	ctx
$\Gamma ::= x_1:T_1, \dots, x_n:T_n$	variable environment

  

(TOP)	(LOOK-UP)	(CTX-C)	(SUM)
$\frac{E_0 \xrightarrow{\bar{D}} E_1 \quad \forall D \in \bar{D}, \bar{D} \vdash D : \text{OK}}{\bar{D} \bar{D}' ID=E_0 \bar{D} \bar{D}' ID=E_1 \bar{D} \bar{D}' DE \rightarrow \bar{D} \bar{D}' DE}$	$\frac{}{t \xrightarrow{\bar{D}} \bar{D}(t)}$	$\frac{E_0 \xrightarrow{\bar{D}} E_1}{\mathcal{E}_c[E_0] \xrightarrow{\bar{D}} \mathcal{E}_c[E_1]}$	$\frac{}{L_1 + L_2 \xrightarrow{\bar{D}} L} L = L_1 + L_2$
(CD-OK)	(TD-OK)	(S-REFL)	
$\frac{C; \bar{D}, C=L_1 \vdash L_1 : \text{OK} \quad L_1 = L_0[\mathbf{This} = C]}{\bar{D} \vdash C=L_0 : \text{OK} \quad \text{coherent}(C, L_1)}$	$\frac{\mathbf{This}; \bar{D}, \mathbf{This}=L \vdash L : \text{OK}}{\bar{D} \vdash t=L : \text{OK}}$	$\frac{}{\bar{D} \vdash T \leq T}$	
(L-OK)	(NESTED-OK)	(SUBTYPE)	
$\frac{\forall M \in \bar{M}, \quad T; \bar{D} \vdash M : \text{OK}}{T; \bar{D} \vdash \{\_ \mathbf{implements} \bar{T} \bar{M}\} : \text{OK}}$	$\frac{T.C; \bar{D} \vdash L : \text{OK}}{T; \bar{D} \vdash C=L : \text{OK}}$	$\frac{\bar{D} \vdash T_2 \leq T_3 \quad \bar{D}(T_1) = \{\_ \mathbf{implements} \bar{T} \_ \}}{\bar{D} \vdash T_1 \leq T_3} T_2 \in \bar{T}$	
(METHOD-OK)			
$\frac{\text{if } e? = e \text{ then } \bar{D}; \Gamma \vdash e : T_0}{T; \bar{D} \vdash \mathbf{static? method} T_0 m(T_1 x_1 \dots T_n x_n) e? : \text{OK}}$			$\begin{array}{l} \text{if } \mathbf{static?} = \mathbf{static} \\ \text{then } \Gamma = x_1 : T_1 \dots x_n : T_n \\ \text{else } \Gamma = \mathbf{this} : T, x_1 : T_1 \dots x_n : T_n \end{array}$
(SUBSUMPTION)	(STATIC-METHOD-CALL)		
$\frac{\bar{D}; \Gamma \vdash e : T_1 \quad \bar{D} \vdash T_1 \leq T_2}{\bar{D}; \Gamma \vdash e : T_2}$	$\frac{\bar{D}; \Gamma \vdash e_1 : T_1 \dots \bar{D}; \Gamma \vdash e_n : T_n}{\bar{D}; \Gamma \vdash T_0.m(e_1 \dots e_n) : T}$		$\mathbf{static method} T m(T_1 x_1 \dots T_n x_n) \_ \in \bar{D}(T_0)$
(x)	(METHOD-CALL)		
$\frac{}{\bar{D}; \Gamma \vdash x : \Gamma(x)}$	$\frac{\bar{D}; \Gamma \vdash e_0 : T_0 \dots \bar{D}; \Gamma \vdash e_n : T_n}{\bar{D}; \Gamma \vdash e_0.m(e_1 \dots e_n) : T}$		$\mathbf{method} T m(T_1 x_1 \dots T_n x_n) \_ \in \bar{D}(T_0)$
(CTXV)	(S-M)	(M)	
$\frac{e_0 \xrightarrow{\bar{D}} e_1}{\mathcal{E}_{\bar{D}}[e_0] \xrightarrow{\bar{D}} \mathcal{E}_{\bar{D}}[e_1]}$	$\frac{}{T.m(\bar{v}_{\bar{D}}) \xrightarrow{\bar{D}} \text{meth}(\bar{D}(T, m), \bar{v}_{\bar{D}})}$	$\frac{}{v_{\bar{D}}.m(\bar{v}_{\bar{D}}) \xrightarrow{\bar{D}} \text{meth}(\bar{D}(T, m), v_{\bar{D}} \bar{v}_{\bar{D}})}$	$v_{\bar{D}} = T.m'(\_)$

■ Figure 1 Formalization

## A.2 Well-formedness

The whole program  $(\overline{DE})$  is well formed if all the traits and classes at top level have unique names. The special class name **This** is not one of those, and the subtype relations are consistent: this means that the implementation of interfaces is not circular, and that  $\forall ID=\mathcal{E}[L] \in \overline{DE}, \text{consistentSubtype}(\overline{DE}, \mathbf{This}=L; L)$ . That is, every literal declares all the methods declared in its super interfaces. The full 42 language allows covariant return types as in Java. Here for simplicity we require them to have the same type declared in the super interface.

**Define**  $\text{consistentSubtype}(\overline{DE}; L)$

•  $\text{consistentSubtype}(\overline{DE}, \{\mathbf{interface?implements} \overline{T} \overline{M}\})$  where

$\forall T \in \overline{T}, \overline{DE}(T) = \{\mathbf{interface} \_ \},$ <sup>9</sup>

$\forall \_ = L \in \overline{M}, \text{consistentSubtype}(\overline{DE}; L)$  and

$\forall m, T \in \overline{T}, \text{if } \mathbf{method} T_0 m(\overline{T}x) \in \overline{DE}(T) \text{ then } \mathbf{method} T_0 m(\overline{T}x) e? \in \overline{M}$

A code literal  $L$  is well formed iff:

- for all methods: parameters have unique names and no parameter is named **this**,
- all methods in a code literal have unique names,
- all nested classes in a code literal have unique names, and no nested class is called **This**,
- all used variables are in scope, and
- all methods in an interface are abstract, and they contain no static methods.

## A.3 Compilation process

The compilation process is particularly interesting, it includes the flattening process and how and when compilation errors may arise. It is composed by rules **TOP**, **LOOK-UP**, **CTX-C** and **SUM**. To model more composition operators, they would each need their own rule.

Rule **TOP** compiles the leftmost top level (trait or class) declaration that needs to be compiled. First it identifies the subset of the program  $\overline{D}$  that can already be typed (second premise). Then the expression is executed under the control of such compiled program (first premise). All the traits inside the expression need to be compiled (rule **LOOK-UP**):  $\forall t, \text{if } E = \mathcal{E}[t] \text{ then } t \in \text{dom}(\overline{D})$ . If the required  $\overline{D}$  cannot be typed, this would cause a compilation error at this stage. Rule **LOOK-UP** replaces a trait name  $t$  with the corresponding literal  $L$ . Since  $\overline{D}$  is all well typed,  $L$  is well typed too. Rule **CTX-C** uses the compilation context to apply a deterministic left to right call by value<sup>10</sup> reduction; thus the leftmost invalid sum that is performed will be the one providing the compilation error.

<sup>9</sup> That is, in this simplified version in order to implement an interface nested in a different top level name, such interface can not be generated using a trait expression. This limitation is lifted in the full language.

<sup>10</sup> In the flattening process, values are code literals  $L$ .

## Separating Use and Reuse to Improve Both

Keeping in mind the order of members in a literal is immaterial, rule `SUM` applies the operator:

**Define**  $L_1 + L_2, \overline{M} + \overline{M}, M + M$

•  $L_1 + L_2 = L_3$  where

$L_1 = \{\mathbf{interface? implements} \overline{T}_1 \overline{M}_1 \overline{M}_0\}$

$L_2 = \{\mathbf{interface? implements} \overline{T}_2 \overline{M}_2 \overline{M}'_0\}$

$L_3 = \{\mathbf{interface? implements} \overline{T}_1, \overline{T}_2 \overline{M}_1, \overline{M}_2 (\overline{M}_0 + \overline{M}'_0)\}$   
 $\text{dom}(\overline{M}_1)$  disjoint  $\text{dom}(\overline{M}_2)$  and  $\text{dom}(\overline{M}_0) = \text{dom}(\overline{M}'_0)$

•  $(M_1 \dots M_n) + (M'_1 \dots M'_n) = (M_1 + M'_1) \dots (M_n + M'_n)$

•  $M_1 + M_2 = M_2 + M_1$

•  $C=L_1 + C=L_2 = C=L_3$  if  $L_1 + L_2 = L_3$

• **static? method**  $T_0 m(\overline{Tx}) + \mathbf{static? method} T_0 m(\overline{Tx})e? = \mathbf{static? method} T_0 m(\overline{Tx})e?$

`Sum` composes the content of the arguments by taking the union of their members and the union of their **implements**. Members with the same name are recursively composed. There are three cases where the composition is impossible.

- *Method-clash*: two methods with the same name are composed, but either their headers have different types or they are both implemented.
- *Class-clash*: a class is composed with an interface.<sup>11</sup>
- *Implements-clash*: the resulting code would not be well formed. For example, in the following `t1+t2` would result in a class `B` implementing `A` with method `a()`, but `B` does not have such method.<sup>12</sup>

```
1 t1={ A= {interface method Void a()} }
2 t2={ A= {interface} B= {implements A} }
```

*Implements-clash* can happen only when composing nested interfaces. Note that while the first two kind of errors are obtained directly by the definition of  $L_1 + L_2$ , *Implements-clash* is obtained from well-formedness, since injecting the resulting  $L$  in to the program would make it ill-formed by  $\text{consistentSubtype}(\overline{DE}, L)$ .

### A.4 Typing

Typing is composed by rules `CD-OK`, `TD-OK`, `L-OK`, `NESTED-OK` and `METHOD-OK`, followed by expression typing rules `SUBSUMPTION`, `METHOD-CALL`, `X` and `STATIC-METHOD-CALL`.

Rules `CD-OK` and `TD-OK` are interesting: a top level class is typed by replacing all occurrences of the name **This** with the class name  $C$ , and is required to be coherent. On the other hand, a top level trait is typed by temporarily adding a mapping for **This** to the typed program.

<sup>11</sup> The full language relaxes this condition, for example an empty class can be seen as an empty interface during composition.

<sup>12</sup> In  $42_\mu$  it could be possible to try to patch class `B`, for example by adding an abstract method `a()`; we choose to instead give an error since in the full  $42$  language such patch would be able to turn coherent private nested classes into abstract (private) ones.

**Define**  $coherent(T, L)$

- $coherent(T, \{\mathbf{interface? implements} \bar{T} \bar{M}\})$  holds where

$$\forall C=L' \in \bar{M} coherent(T.C, L')$$

and either  $\mathbf{interface?} = \mathbf{interface}$

$$\text{or } \forall \mathbf{method} T' m(\bar{T}x) \in \bar{M}, \text{state}(\text{factory}(T, \bar{M}), \mathbf{method} T' m(\bar{T}x))$$

A Literal is *coherent* if all the nested classes are coherent, and either the Literal is an interface, there are no static methods, or all the static methods are a valid *state* method of the candidate *factory*. Note, by asking for  $\mathbf{method} T' m(\bar{T}x) \in \bar{M}$  we select only abstract methods.

**Define**  $factory(T, \bar{M})$

- $factory(T, M_1 \dots M_n) = M_i = \mathbf{static method} T m(\_)$  where

$$\forall j \neq i. M_j \text{ is not of the form } \mathbf{static method} \_ \_ (\_)$$

The factory is the only static abstract method, and its return type is the nominal type of our class.

**Define**  $state(M, M')$

- $\text{state}(\mathbf{static method} T m(T_1 x_1 \dots T_n x_n), \mathbf{method} T_i x_i(\_))$
- $\text{state}(\mathbf{static method} T m(T_1 x_1 \dots T_n x_n), \mathbf{method} T \text{ with } x_i(T_i \text{ that}))$

A non static method is part of the *abstract state* if it is a valid getter or wither. In this simple formalism without imperative features we do not offer setters.

Rule NESTED-OK helps to accumulate the type of **this** so that rule METHOD-OK can use it. Rule L-OK is so simple since all the checks related to correctly implementing interfaces are delegated to the well formedness criteria. The expression typing rules are straightforward and standard.

## A.5 Formal properties

As can be expected,  $42_\mu$  ensures conventional soundness of expression reduction. This property is expressed on a completely flattened program (a program where all  $E$  are of form  $L$ ):

**Theorem A.1** (Main Soundness). *if  $\bar{D} \vdash OK$ ,  $e$  not of form  $v_{\bar{b}}$  and  $\bar{D} \vdash e : T$  then  $e \leftarrow_{\bar{D}} \_$*

The proof is standard since the flattened language is just a minor variation over FJ.

In addition to conventional soundness of expression reduction,  $42_\mu$  ensures soundness of the compilation process itself. A similar property was called meta-level-soundness in [37]; here we can obtain the same result in a much simpler setting. We denote  $wrong(\bar{D}, E)$  to be the number of  $L$ s such that  $E = \mathcal{E}[L]$  and not  $\bar{D} \vdash L : OK$ .

**Theorem A.2** (Compilation Soundness). *if  $E_0 \xrightarrow{\bar{D}} E_1$  then  $wrong(\bar{D}, E_0) \geq wrong(\bar{D}, E_1)$ .*

This can be proved by cases on the applied reduction rule:

- LOOK-UP preserves the number of wrong literals:  $t \in \bar{D}$  and  $\bar{D}$  is well typed by TOP preconditions.
- SUM either preserves or reduces the number of wrong literals: the core of the proof is to show that the sum of two well typed literals produces a well typed one. A code literal is well typed (L-OK) if all its method bodies are correct. This holds since those same method bodies are well typed in a strictly weaker environment with

## Separating Use and Reuse to Improve Both

respect to the one used to type the result. This is because every member in the result of the sum is structurally a subtype of the corresponding members in the operands. Note that by well formedness, if `SUM` is applied the result still respect *consistentSubtype*.

### Compilation Soundness has two important corollaries:

- A class declared without literals is well-typed after flattening; no need of further checking.
- If a class is declared by using literals  $L_1 \dots L_n$ , and after successful flattening  $C = L$  can not be type-checked, then the issue was originally present in one of  $L_1 \dots L_n$ . This also means that as an optimization strategy we may remember what method bodies come from traits and what method bodies come from code literals, and only type-check the latter. If the result can not be type-checked, either it is intrinsically illtyped or a referred type is declared *after* the current class. As we see in the next section, we leverage on this to allow recursive types.

## A.6 Advantages of our compilation process

Our typing discipline is very simple from a formal perspective, and is what distinguishes our approach from a simple minded code composition macros [4] or rigid module composition [1]. It is built on two core ideas:

**1: Traits are *well-typed* before being reused.** For example:

```
1 t={method int m(){return 2;}}
2   method int n(){return this.m()+1;}}
```

`t` is well typed since `m()` is declared inside of `t`, while the following would be illtyped:

```
1 t1={method int n(){return this.m()+1;}} //illtyped
```

**2: Code literals are *not* required to be well-typed before flattening.**

A literal  $L$  in a declaration  $D$  must be well formed and respect *consistentSubtype*, but it is not type-checked until flattening is complete: only the result is required to be well-typed. For example the following is correct since the result of the flattening is well-typed:

```
1 C= Use t, {method int k(){return this.n()+this.m();}}//correct code
```

The code literal `{method int k(){ return this.n()+this.m();}}` is not well typed: `n`, `m` are not locally defined. This code would fail in many similar works in literature [5, 7, 16] where the literals have to be *self contained*. In this case we would have been forced to declare abstract methods `n` and `m`, even if `t` already provides such methods.

This relaxation allows multiple declarations to be flattened one at the time, without typing them individually, and only typing them all together. In this way, we support



recursive types<sup>13</sup> between multiple class declarations without the need of predicting the resulting shape.<sup>14</sup>

As seen in TOP, our compilation process proceeds in a top-down fashion, flattening one declaration at a time, a declaration needs to be type-checked where their type is first needed, that is, when they are required to type a trait used in a code expression. That is, in  $42_\mu$  typing and flattening are interleaved. We assume our compilation process stops as soon as an error arises. For example:

```

1 ta={method int ma(){return 2;}}
2 tc={method int mc(A a, B b){return b.mb(a);}}
3 A= Use ta
4 B= {method int mb(A a){return a.ma()+1;}}
5 C= Use tc, {method int hello(){return 1;}}

```

In this scenario, since we compile top down, we first need to generate A. To generate A, we need to use ta (but we do not need tc, in rule TOP,  $\bar{D} = ta$  and  $\bar{D}' = tc$ ). At this moment, tc cannot be compiled/checked alone: information about A and B is needed. To modularly ensure well-typedness, we only require ta to be well typed at this stage; if it is not a type-error will be raised immediately. Now, we need to generate c, and hence type-check tc. A is guaranteed to be already type-checked (since it is generated by an expression that does not contain any L), and B can be typed. Finally tc can be typed and reused. If the SUM rule could not be performed (for example if tc had a method hello too) a composition error would be generated at this stage. On the other hand, if B and C were swapped, as in:

```

1 C= Use tc, {method int hello(){return 1;}}
2 B= {method int mb(A a){return a.ma()+1;}}

```

we would be unable to type tc, since we need to know the structure of A and B. A type error would be generated.

### The cost: what expressive power we lose

We require declarations to be provided in the right dependency order, but sometimes no such order exists. An example of a “morally correct” program where no right order exists is the following:

```

1 t= { int mt(A a){return a.ma();}}
2 A= Use t, {int ma(){return 1;}}

```

Here the correctness of t depends on A, that is in turn generated using t. We believe any typing allowing such programs would be fragile with respect to code evolution, and could make human understanding of the code-reuse process much harder. In sharp contrast with others (TR, PT, DJ, but also Java, C#, and Scala) we chose to not support this kind of involved programs.

TR, PT, DJ, Java, C#, and Scala accept a great deal of complexity in order to predict the structural shape of the resulting code before doing the actual code reuse/adapta-

<sup>13</sup> OO languages leverage on recursive types most of the times: for example String may offer a `Int size()` method, and Int may offer a `String toString()` method. This means that typing classes String and Int in isolation one at a time is not possible.

<sup>14</sup> This is needed in full 42: it is impossible to predict the resulting shape since arbitrary code can run at compile time.

## Separating Use and Reuse to Improve Both

tion. Those approaches logically divide the program in groups of mutually dependent classes, where each group may depend on a number of other groups. This forms a direct acyclic graph of groups. To type a group, all depended groups are typed, then the signature/structural shape of all the classes of the group are extracted. Finally, with the information of the dependent groups and the current group, it is possible to type-check the implementation of each class in the group.

In the world of strongly typed languages we are tempted to first check that all will go well, and then perform the flattening. Such methodology would be redundant in our setting: we can only reuse code through trait names; but our point of relaxation is only the code literal: in no way can an error “move around” and be duplicated during the compilation process. That is, our approach allows safe libraries of traits and classes to be typechecked once, and then deployed and reused by multiple clients: as Theorem A.2 states, in  $42_\mu$  no type error will emerge from library code.

### A.7 Expression reduction

Our reduction rules are incredibly simple and standard. A great advantage of our compilation model is that expressions are executed on a simple fully flattened program, where all the composition operators have been removed. From the point of view of expression reduction,  $42_\mu$  is a simple language of interfaces and final classes, where nested classes give structure to code but have no special semantics. The reduction of expressions is defined by rules  $\text{ctx-v}$ ,  $\text{s-m}$ , and  $\text{m}$ . The only interesting point is the auxiliary function *meth*:

**Define**  $\text{meth}(M, \bar{v}_D)$

- $\text{meth}(\text{static method } T \ m(T_1 x_1 \dots T_n x_n) \ e, v_{\bar{D}1} \dots v_{\bar{D}n}) = e[x_1 = v_{\bar{D}1}, \dots, e_n = v_{\bar{D}n}]$
- $\text{meth}(\text{method } T \ m(T_1 x_1 \dots T_n x_n) \ e, v_{\bar{D}0}, \dots, v_{\bar{D}n}) = e[\text{this} = v_{\bar{D}0}, x_1 = v_{\bar{D}1}, \dots, e_n = v_{\bar{D}n}]$
- $\text{meth}(\text{method } T_i \ x_i(), T.m(v_{\bar{D}1} \dots v_{\bar{D}n})) = v_{\bar{D}i}$   
 where  $\bar{D}(T, m) = \text{static method } T \ m(T_1 x_1 \dots T_n x_n)$
- $\text{meth}(\text{method } T \ \text{with } x_i(T_i \ \text{that}), T.m(v_{\bar{D}1} \dots v_{\bar{D}n}) \ v_{\bar{D}}) = T.m(v_{\bar{D}1} \dots v_{\bar{D}i-1}, v_{\bar{D}}, v_{\bar{D}i+1} \dots v_{\bar{D}n})$   
 where  $\bar{D}(T, m) = \text{static method } T \ m(T_1 x_1 \dots T_n x_n)$

Here we take care of reading method bodies and preparing for execution. The first case is for static methods and the second is for instance methods. The third and fourth cases are more interesting, since they take care of the abstract state: the third case reduce getters and the fourth reduces withers. In our formalisation we are not modelling state mutation, so there is no case for setters.

We omit the proof of conventional soundness for the reduction. It is unsurprising, since the flattened calculus is like a simplified version of Featherweight Java [23].

## **B** Benefits without heavy costs

In this section we discuss why we believe the design of  $42_\mu$  is not intrinsically harder to use than Java. This appendix is not fundamental to understanding the main content of the paper, but may clarify misconceptions.

## B.1 Introducing more names

At first it may seem that our approach requires introducing and keeping more names in mind, as for the `set` and `Set` example; however this is not the case. The *user* of `Set` needs to keep in mind only the class `Set`. The *re-user* of `set` needs to keep in mind only the trait `set`. Those are quite different roles, so they may very well be handled by different programmers. As shown in the `set/Set` example, defining a reusable class takes only an extra line with respect to defining a final one. In Java the default is “non final”, and the programmer can write `final` to prevent code reuse, while in 42 the default is reversed.

It could be possible to define syntax sugar, like this:

```
1 reusable Set={...}
2 //expands into
3 set={...}
4 Set=set
```

While this may sound appealing to Java programmers, it only saves one line, and it may obscure the actual behaviour of the code. This kind of behaviour is offered in Cecil [14], where it makes sense, since in Cecil classes are not types. Cecil offers syntax sugar to declare classes and interfaces with the same structural type in a single declaration.

## B.2 Two ways to separate subclassing and subtyping

There are two different ways to separate subclassing and subtyping:

- (1) *Classes are not types*: in this solution (used by Cecil, Toil/PolyToil, and most structurally typed languages) classes and types are different concepts; thus subclassing is not subtyping. This also means that every method call is dynamically dispatched. Some authors consider this to be a security problem, and it is out of those concerns that Java designers decided to support `final` classes.<sup>15</sup>
- (2) *Classes are not extensible*: this is our solution; classes are types, but can not be reused. This requires introducing another concept for code reuse: traits. We think our solution is much less radical than the former one. A key advantage is that all class types are exact types, thus method calls on them are statically dispatched; supporting easier reasoning and higher security and efficiency. On the other hand, interface types are not exact; obviously method calls on interface types are dynamically dispatched, since there is no implementation. This builds towards a simpler mental model, and prevents bugs where programmers may

<sup>15</sup> For example, a Java programmer may assume the call `myPoint.getX()` to behave as specified in the `Point` class. However, dynamic class loading may be used in a malicious way to provide a `Point` instance with dangerous behaviour. This can easily be prevented by declaring `Point` `final`. To insist that all calls be dynamically dispatched requires disabling dynamic class loading or verifying all dynamically loaded code.

## Separating Use and Reuse to Improve Both

forget about dynamic dispatch and reasoning over method calls using the specific implementation found in the receivers static type.

Both ways enforce essentially the same solution for the ‘this-leaking’ problem:

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

In (1) the error is using A as a type in `Utils.m`, while in (2) the error is using `extends`.

The solution proposed by (1) would look like this:

```
1 type IA{interface method int ma()}//type = interface
2 class Utils{static method int m(IA a){return ...} }
3 class A{implements IA
4   method int ma(){return Utils.m(this);}}
5 class B extends A{ int mb(){return this.ma();} }
```

while our solution (2) is just one line longer:

```
1 IA={interface method int ma()}//interface with abstract method
2 Utils={static method int m(IA a){return ...} }
3 ta={implements IA
4   method int ma(){return Utils.m(this);}}
5 A=Use ta
6 B=Use ta,{ int mb(){return this.ma();} }
```

### B.3 Class hierarchies

By having all classes final, we prevent class hierarchies. However, we allow both trait hierarchies and interface hierarchies. We believe having both these hierarchies, one for code reuse, and one for subtyping, can improve code maintainability, since these two hierarchies will be able to evolve independently. Furthermore, having two hierarchies does not makes code comprehension harder, since as for before, they have different roles and thus are relevant only one at the time.

Often, class hierarchies are used as a hack in order to obtain code reuse (and in those cases, the LSP is often violated). In  $42_\mu$ , trait hierarchies will serve the same role better.

### B.4 Constructors and initialization

Java allows constructors to perform arbitrary operations. The same behaviour can be obtained in  $42_\mu$  by using a factory method, like so:

```
1 class A{
2   public int x;
3   public A(String s){
4     this.x=/*any computation here*/;
5   }
6 }
```

could be expressed in  $42_\mu$  as

```

1 A={
2   method int x();
3   static method This of(int x)
4   static method This of(String s){
5     return This.of(/*any computation here*/);
6   }
7 }

```

For space reasons we have not yet explained how to allow traits to have their own state with a default initialization. This could be solved by adding a default initializer to the abstract state operations (getters, setters, withers). For example:

```

1 t={
2   method int x();
3   method void x(int x);
4   method int defaultX(){return 0;}//new concept here!
5 }

```

would loosely correspond to a Java abstract class:

```

1 abstract class T{
2   private int x=0;
3   public int x(){return x;}
4   public void x(int that){return this.x=that;}
5 }

```

and could be used to obtain concrete classes where the abstract factory does not need to initialize the field  $x$ . For example the following classes would be coherent:

```

1 A= Use t,{ static method This of()}
2 B= Use t,{ method String s(); static method This of(String s)}
3 ...
4 A a=A.of();
5 a.x()==0;//assert a.x()==0
6 a.x(42);
7 a.x()==42;//assert a.x()==42

```

Finally, when self instantiation is not needed, declaring the abstract factory can be made redundant, as is done in the full 42 language.

Moreover, when code reuse is involved, 42<sub>μ</sub> is much simpler to use than Java, C++ or C#: a class extending another class usually needs to define a constructor calling a super constructor. This typically has as many parameters as the full set of fields. Those parameters are then used at least one time in the body of the constructor. In 42<sub>μ</sub>, to compose traits without adding fields, there is no need to repeat the abstract factory. To add a field, you need to define a new abstract factory and implement the "super" factory method by calling the new one. Thus, when adding a field the amount of code between 42 and Java is about the same, but is much easier when no fields are added.

## B.5 Self instantiation is very useful

In many cases the absence of self instantiation causes bugs: often programmers “think” that `new c()` inside of class `c` would create an instance of the “current class”, while of course `class D extends c` still creates instances of `c`. The absence of self instantiation

## Separating Use and Reuse to Improve Both

also prevents using functional programming patterns in OO languages. Just looking at how tedious it is to implement an immutable `Point` class in Java with `withers`, it is sufficient to dissuade most programmers.

### B.6 Comprehensibility and usability

Of course it could take a while to adapt from the Java model to the  $42_{\mu}$  one. However, when it comes to model complex reuse scenarios, we believe  $42_{\mu}$  is much more explicit and clear than Scala and other options.

In particular, with respect to Java/Scala we encourage more explicit code since:

- Sub-typing needs to be explicitly declared.
- When sub-typing is needed, an interface needs to be declared. Thus, when using a class type, it is always clear what implementation is referred to.
- $42_{\mu}$  requires being explicit when subtyping and subclassing are both desired by requiring an extra interface.
- **this** is required to access fields/methods, so they can not be confused with local variables.
- Our runtime semantics, as in untyped languages, do not depend on any static type information.

The last point is quite important: languages where inheritance does not induce subtyping that were proposed in the past tend to have more complex object models, instead our flattened language is just a language of interfaces and final classes. Novice programmers may start by learning that. They could learn to use traits later, and define their own traits even later.

## C Case studies complete code

Here, for reference, you can find the complete code of the case studies. It is quite long, we report it here to help reproducibility of our results. The interested reader can navigate it in order to gain a better understanding on how we encoded the various solutions in the various styles and languages, but is not needed to understand the overall value of our paper.

While evaluating our approach, we consider not only lines of code but also the number of methods and classes. Our particular examples are focused on code reuse, thus the method implementations are all trivial. We believe that in this particular context the number of methods/classes is a good indicator.

### C.1 Point algebra

#### C.1.1 Java7

```
1 //case 0 of 16
2 class Point { //6 lines
3     final int x;
4     final int y;
5
6     public Point(int x, int y) {
```

```

7   this.x = x;
8   this.y = y;
9   }
10  }
11
12 // 1
13 class PointSum extends Point { //5*4
14     public PointSum(int x, int y) {
15         super(x, y);
16     }
17
18     public PointSum sum(Point that) {
19         return new PointSum(this.x + that.x, this.y + that.y);
20     }
21 }
22
23 // 2
24 class PointSub extends Point {
25     public PointSub(int x, int y) {
26         super(x, y);
27     }
28
29     public PointSub sub(Point that) {
30         return new PointSub(this.x - that.x, this.y - that.y);
31     }
32 }
33
34 // 3
35 class PointMul extends Point {
36     public PointMul(int x, int y) {
37         super(x, y);
38     }
39
40     public PointMul mul(Point that) {
41         return new PointMul(this.x * that.x, this.y * that.y);
42     }
43 }
44
45 // 4
46 class PointDiv extends Point {
47     public PointDiv(int x, int y) {
48         super(x, y);
49     }
50
51     public PointDiv div(Point that) {
52         return new PointDiv(this.x / that.x, this.y / that.y);
53     }
54 }
55
56 // 5
57 class PointSumSub extends PointSum { //7*6
58     public PointSumSub(int x, int y) {
59         super(x, y);
60     }
61
62     public PointSumSub sum(Point that) {
63         return new PointSumSub(this.x + that.x, this.y + that.y);
64     }
65
66     public PointSumSub sub(Point that) {
67         return new PointSumSub(this.x - that.x, this.y - that.y);
68     }
69 }
70
71 // 6
72 class PointSumMul extends PointSum {
73     public PointSumMul(int x, int y) {
74         super(x, y);
75     }
76
77     public PointSumMul sum(Point that) {
78         return new PointSumMul(this.x + that.x, this.y + that.y);
79     }
80
81     public PointSumMul mul(Point that) {
82         return new PointSumMul(this.x * that.x, this.y * that.y);
83     }
84 }
85
86 // 7
87 class PointSumDiv extends PointSum {
88     public PointSumDiv(int x, int y) {
89         super(x, y);
90     }
91
92     public PointSumDiv sum(Point that) {
93         return new PointSumDiv(this.x + that.x, this.y + that.y);
94     }
95
96     public PointSumDiv div(Point that) {
97         return new PointSumDiv(this.x / that.x, this.y / that.y);

```

## Separating Use and Reuse to Improve Both

```
98 }
99 }
100
101 // 8
102 class PointSubMul extends PointSub {
103     public PointSubMul(int x, int y) {
104         super(x, y);
105     }
106
107     public PointSubMul sub(Point that) {
108         return new PointSubMul(this.x - that.x, this.y - that.y);
109     }
110
111     public PointSubMul mul(Point that) {
112         return new PointSubMul(this.x * that.x, this.y * that.y);
113     }
114 }
115
116 // 9
117 class PointSubDiv extends PointSub {
118     public PointSubDiv(int x, int y) {
119         super(x, y);
120     }
121
122     public PointSubDiv sub(Point that) {
123         return new PointSubDiv(this.x - that.x, this.y - that.y);
124     }
125
126     public PointSubDiv div(Point that) {
127         return new PointSubDiv(this.x / that.x, this.y / that.y);
128     }
129 }
130
131 // 10
132 class PointMulDiv extends PointMul {
133     public PointMulDiv(int x, int y) {
134         super(x, y);
135     }
136
137     public PointMulDiv mul(Point that) {
138         return new PointMulDiv(this.x * that.x, this.y * that.y);
139     }
140
141     public PointMulDiv div(Point that) {
142         return new PointMulDiv(this.x / that.x, this.y / that.y);
143     }
144 }
145
146 // 11
147 class PointSumSubDiv extends PointSumSub { //9*4
148     public PointSumSubDiv(int x, int y) {
149         super(x, y);
150     }
151
152     public PointSumSubDiv sum(Point that) {
153         return new PointSumSubDiv(this.x + that.x, this.y + that.y);
154     }
155
156     public PointSumSubDiv sub(Point that) {
157         return new PointSumSubDiv(this.x - that.x, this.y - that.y);
158     }
159
160     public PointSumSubDiv div(Point that) {
161         return new PointSumSubDiv(this.x / that.x, this.y / that.y);
162     }
163 }
164
165 // 12
166 class PointSumSubMul extends PointSumSub {
167     public PointSumSubMul(int x, int y) {
168         super(x, y);
169     }
170
171     public PointSumSubMul sum(Point that) {
172         return new PointSumSubMul(this.x + that.x, this.y + that.y);
173     }
174
175     public PointSumSubMul sub(Point that) {
176         return new PointSumSubMul(this.x - that.x, this.y - that.y);
177     }
178
179     public PointSumSubMul mul(Point that) {
180         return new PointSumSubMul(this.x * that.x, this.y * that.y);
181     }
182 }
183
184 // 13
185 class PointSumMulDiv extends PointMulDiv {
186     public PointSumMulDiv(int x, int y) {
187         super(x, y);
188     }
```



```

189
190 public PointSumMulDiv sum(Point that) {
191     return new PointSumMulDiv(this.x + that.x, this.y + that.y);
192 }
193
194 public PointSumMulDiv mul(Point that) {
195     return new PointSumMulDiv(this.x * that.x, this.y * that.y);
196 }
197
198 public PointSumMulDiv div(Point that) {
199     return new PointSumMulDiv(this.x / that.x, this.y / that.y);
200 }
201 }
202
203 // 14
204 class PointSubMulDiv extends PointMulDiv {
205     public PointSubMulDiv(int x, int y) {
206         super(x, y);
207     }
208
209     public PointSubMulDiv sub(Point that) {
210         return new PointSubMulDiv(this.x - that.x, this.y - that.y);
211     }
212
213     public PointSubMulDiv mul(Point that) {
214         return new PointSubMulDiv(this.x * that.x, this.y * that.y);
215     }
216
217     public PointSubMulDiv div(Point that) {
218         return new PointSubMulDiv(this.x / that.x, this.y / that.y);
219     }
220 }
221
222 // 15
223 class PointSumSubMulDiv extends PointSumSubMul { //11
224     public PointSumSubMulDiv(int x, int y) {
225         super(x, y);
226     }
227
228     public PointSumSubMulDiv sum(Point that) {
229         return new PointSumSubMulDiv(this.x + that.x, this.y + that.y);
230     }
231
232     public PointSumSubMulDiv sub(Point that) {
233         return new PointSumSubMulDiv(this.x - that.x, this.y - that.y);
234     }
235
236     public PointSumSubMulDiv mul(Point that) {
237         return new PointSumSubMulDiv(this.x * that.x, this.y * that.y);
238     }
239
240     public PointSumSubMulDiv div(Point that) {
241         return new PointSumSubMulDiv(this.x / that.x, this.y / that.y);
242     }
243 }

```

## C.1.2 Classless Java

```

1 // 0
2 @Obj // 3 lines
3 interface Point {
4     static Point of(int x, int y) { // "of" methods are generated by @Obj thus their lines are not counted
5         return new Point() {
6             public int x() {
7                 return x;
8             }
9
10            public int y() {
11                return y;
12            }
13        };
14    }
15
16    int x();
17
18    int y();
19 }
20
21 // 1
22 @Obj // 3*4 lines
23 interface PointSum extends Point {
24     static PointSum of(int x, int y) {
25         return new PointSum() {
26             public int x() {
27                 return x;
28             }
29
30            public int y() {

```

## Separating Use and Reuse to Improve Both

```
31     return y;
32     };
33 };
34 }
35
36 default PointSum sum(Point that) {
37     return of(this.x() + that.x(), this.y() + that.y());
38 }
39 }
40
41 // 2
42 @Obj
43 interface PointSub extends Point {
44     static PointSub of(int x, int y) {
45         return new PointSub() {
46             public int x() {
47                 return x;
48             }
49
50             public int y() {
51                 return y;
52             }
53 };
54 }
55
56 default PointSub sub(Point that) {
57     return of(this.x() - that.x(), this.y() - that.y());
58 }
59 }
60
61 // 3
62 @Obj
63 interface PointMul extends Point {
64     static PointMul of(int x, int y) {
65         return new PointMul() {
66             public int x() {
67                 return x;
68             }
69
70             public int y() {
71                 return y;
72             }
73 };
74 }
75
76 default PointMul mul(Point that) {
77     return of(this.x() * that.x(), this.y() * that.y());
78 }
79 }
80
81 // 4
82 @Obj
83 interface PointDiv extends Point {
84     static PointDiv of(int x, int y) {
85         return new PointDiv() {
86             public int x() {
87                 return x;
88             }
89
90             public int y() {
91                 return y;
92             }
93 };
94 }
95
96 default PointDiv div(Point that) {
97     return of(this.x() / that.x(), this.y() / that.y());
98 }
99 }
100
101 // 5
102 @Obj // 5*6 lines
103 interface PointSumMul extends PointSum, PointMul {
104     static PointSumMul of(int x, int y) {
105         return new PointSumMul() {
106             public int x() {
107                 return x;
108             }
109
110             public int y() {
111                 return y;
112             }
113 };
114 }
115
116 default PointSumMul sum(Point that) { // we have to rewrite the method to call the new "of"
117     return of(this.x() + that.x(), this.y() + that.y()); // in order to produce an instance of PointSumMul
118 }
119
120 default PointSumMul mul(Point that) {
121     return of(this.x() * that.x(), this.y() * that.y());
122 }
```

```

122 }
123 }
124
125 // 6
126 @Obj
127 interface PointSumSub extends PointSum, PointSub {
128     static PointSumSub of(int x, int y) {
129         return new PointSumSub() {
130             public int x() {
131                 return x;
132             }
133
134             public int y() {
135                 return y;
136             }
137         };
138     }
139
140     default PointSumSub sum(Point that) { // we have to rewrite the method to call the new "of"
141         return of(this.x() + that.x(), this.y() + that.y()); // in order to produce an instance of PointSumMul
142     }
143
144     default PointSumSub sub(Point that) {
145         return of(this.x() - that.x(), this.y() - that.y());
146     }
147 }
148
149 // 7
150 @Obj
151 interface PointSumDiv extends PointSum, PointDiv {
152     static PointSumDiv of(int x, int y) {
153         return new PointSumDiv() {
154             public int x() {
155                 return x;
156             }
157
158             public int y() {
159                 return y;
160             }
161         };
162     }
163
164     default PointSumDiv sum(Point that) { // we have to rewrite the method to call the new "of"
165         return of(this.x() + that.x(), this.y() + that.y()); // in order to produce an instance of PointSumMul
166     }
167
168     default PointSumDiv div(Point that) {
169         return of(this.x() / that.x(), this.y() / that.y());
170     }
171 }
172
173 // 8
174 @Obj
175 interface PointSubMul extends PointMul, PointSub {
176     static PointSubMul of(int x, int y) {
177         return new PointSubMul() {
178             public int x() {
179                 return x;
180             }
181
182             public int y() {
183                 return y;
184             }
185         };
186     }
187
188     default PointSubMul mul(Point that) { // we have to rewrite the method to call the new "of"
189         return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
190     }
191
192     default PointSubMul sub(Point that) {
193         return of(this.x() - that.x(), this.y() - that.y());
194     }
195 }
196
197 // 9
198 @Obj
199 interface PointSubDiv extends PointSub, PointDiv {
200     static PointSubDiv of(int x, int y) {
201         return new PointSubDiv() {
202             public int x() {
203                 return x;
204             }
205
206             public int y() {
207                 return y;
208             }
209         };
210     }
211
212     default PointSubDiv sub(Point that) { // we have to rewrite the method to call the new "of"

```

## Separating Use and Reuse to Improve Both

```
213     return of(this.x() - that.x(), this.y() - that.y()); // in order to produce an instance of PointSumMul
214 }
215
216 default PointSubDiv div(Point that) {
217     return of(this.x() / that.x(), this.y() / that.y());
218 }
219 }
220
221 // 10
222 @Obj
223 interface PointMulDiv extends PointMul, PointDiv {
224     static PointMulDiv of(int x, int y) {
225         return new PointMulDiv() {
226             public int x() {
227                 return x;
228             }
229
230             public int y() {
231                 return y;
232             }
233         };
234     }
235
236 default PointMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
237     return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
238 }
239
240 default PointMulDiv div(Point that) {
241     return of(this.x() / that.x(), this.y() / that.y());
242 }
243 }
244
245 // 11
246 @Obj // 7*4 lines
247 interface PointSumSubDiv extends PointSumSub, PointSumDiv, PointSubDiv {
248     static PointSumSubDiv of(int x, int y) {
249         return new PointSumSubDiv() {
250             public int x() {
251                 return x;
252             }
253
254             public int y() {
255                 return y;
256             }
257         };
258     }
259
260 default PointSumSubDiv sum(Point that) {
261     return of(this.x() + that.x(), this.y() + that.y());
262 }
263
264 default PointSumSubDiv sub(Point that) {
265     return of(this.x() - that.x(), this.y() - that.y());
266 }
267
268 default PointSumSubDiv div(Point that) {
269     return of(this.x() / that.x(), this.y() / that.y());
270 }
271 }
272
273 // 12
274 @Obj
275 interface PointSumSubMul extends PointSumSub, PointSumMul, PointSubMul {
276     static PointSumSubMul of(int x, int y) {
277         return new PointSumSubMul() {
278             public int x() {
279                 return x;
280             }
281
282             public int y() {
283                 return y;
284             }
285         };
286     }
287
288 default PointSumSubMul sum(Point that) {
289     return of(this.x() + that.x(), this.y() + that.y());
290 }
291
292 default PointSumSubMul sub(Point that) {
293     return of(this.x() - that.x(), this.y() - that.y());
294 }
295
296 default PointSumSubMul mul(Point that) {
297     return of(this.x() * that.x(), this.y() * that.y());
298 }
299 }
300
301 // 13
302 @Obj
303 interface PointSumMulDiv extends PointSumMul, PointMulDiv, PointSumDiv {
```

```

304 static PointSumMulDiv of(int x, int y) {
305     return new PointSumMulDiv() {
306         public int x() {
307             return x;
308         }
309
310         public int y() {
311             return y;
312         }
313     };
314 }
315
316 default PointSumMulDiv sum(Point that) {
317     return of(this.x() + that.x(), this.y() + that.y());
318 }
319
320 default PointSumMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
321     return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
322 }
323
324 default PointSumMulDiv div(Point that) {
325     return of(this.x() / that.x(), this.y() / that.y());
326 }
327 }
328
329 // 14
330 @Obj
331 interface PointSubMulDiv extends PointSubMul, PointMulDiv, PointSubDiv {
332     static PointSubMulDiv of(int x, int y) {
333         return new PointSubMulDiv() {
334             public int x() {
335                 return x;
336             }
337
338             public int y() {
339                 return y;
340             }
341         };
342     }
343
344     default PointSubMulDiv sub(Point that) {
345         return of(this.x() - that.x(), this.y() - that.y());
346     }
347
348     default PointSubMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
349         return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
350     }
351
352     default PointSubMulDiv div(Point that) {
353         return of(this.x() / that.x(), this.y() / that.y());
354     }
355 }
356
357 // 15
358 @Obj // 9*1 lines
359 interface PointSumSubMulDiv extends PointSumSubMul, PointSumMulDiv, PointSumSubDiv, PointSubMulDiv {
360     static PointSumSubMulDiv of(int x, int y) {
361         return new PointSumSubMulDiv() {
362             public int x() {
363                 return x;
364             }
365
366             public int y() {
367                 return y;
368             }
369         };
370     }
371
372     default PointSumSubMulDiv sum(Point that) {
373         return of(this.x() + that.x(), this.y() + that.y());
374     }
375
376     default PointSumSubMulDiv sub(Point that) {
377         return of(this.x() - that.x(), this.y() - that.y());
378     }
379
380     default PointSumSubMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
381         return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
382     }
383
384     default PointSumSubMulDiv div(Point that) {
385         return of(this.x() / that.x(), this.y() / that.y());
386     }
387 }

```

### C.1.3 Scala

```

1 trait tPointState { // 5

```

## Separating Use and Reuse to Improve Both

```
2  type p <: tPointState
3  def x: Int
4  def y: Int
5  def of(x:Int,y:Int):p
6  }
7
8  trait tPointSum extends tPointState { //3*4
9  def sum(that:p)=
10   this.of(this.x+that.x,this.y+that.y)
11 }
12
13 trait tPointSub extends tPointState {
14 def sub(that:p)=
15   this.of(this.x-that.x,this.y-that.y)
16 }
17
18 trait tPointMul extends tPointState {
19 def mul(that:p)=
20   this.of(this.x*that.x,this.y*that.y)
21 }
22
23 trait tPointDiv extends tPointState {
24 def div(that:p)=
25   this.of(this.x/that.x,this.y/that.y)
26 }
27 //glue code from now on
28 class Point(val x:Int, val y:Int) extends tPointState { //4*16
29 override type p = Point
30 override def of(x:Int,y:Int)=
31   new Point(x,y)// I have to repeat the name of the class 3 times
32 }
33
34 class PointSum(val x:Int, val y:Int) extends tPointSum {
35 override type p = PointSum
36 override def of(x:Int,y:Int)=
37   new PointSum(x,y)
38 }
39
40 class PointSub(val x:Int, val y:Int) extends tPointSub {
41 override type p = PointSub
42 override def of(x:Int,y:Int)=
43   new PointSub(x,y)
44 }
45
46 class PointMul(val x:Int, val y:Int) extends tPointMul {
47 override type p = PointMul
48 override def of(x:Int,y:Int)=
49   new PointMul(x,y)
50 }
51
52 class PointDiv(val x:Int, val y:Int) extends tPointDiv {
53 override type p = PointDiv
54 override def of(x:Int,y:Int)=
55   new PointDiv(x,y)
56 }
57
58 class PointSumSub(val x:Int, val y:Int) extends tPointSum with tPointSub {
59 override type p = PointSumSub
60 override def of(x:Int,y:Int)=
61   new PointSumSub(x,y)
62 }
63
64 class PointSumMul(val x:Int, val y:Int) extends tPointSum with tPointMul {
65 override type p = PointSumMul
66 override def of(x:Int,y:Int)=
67   new PointSumMul(x,y)
68 }
69
70 class PointSumDiv(val x:Int, val y:Int) extends tPointSum with tPointDiv {
71 override type p = PointSumDiv
72 override def of(x:Int,y:Int)=
73   new PointSumDiv(x,y)
74 }
75
76 class PointSubMul(val x:Int, val y:Int) extends tPointSub with tPointMul {
77 override type p = PointSubMul
78 override def of(x:Int,y:Int)=
79   new PointSubMul(x,y)
80 }
81
82 class PointSubDiv(val x:Int, val y:Int) extends tPointSub with tPointDiv {
83 override type p = PointSubDiv
84 override def of(x:Int,y:Int)=
85   new PointSubDiv(x,y)
86 }
87
88 class PointMulDiv(val x:Int, val y:Int) extends tPointMul with tPointDiv {
89 override type p = PointMulDiv
90 override def of(x:Int,y:Int)=
91   new PointMulDiv(x,y)
92 }
```

```

93
94 class PointSumSubDiv(val x:Int, val y:Int) extends tPointSum with tPointSub with tPointDiv {
95   override type p = PointSumSubDiv
96   override def of(x:Int,y:Int)=
97     new PointSumSubDiv(x,y)
98 }
99
100 class PointSumSubMul(val x:Int, val y:Int) extends tPointSum with tPointSub with tPointMul {
101   override type p = PointSumSubMul
102   override def of(x:Int,y:Int)=
103     new PointSumSubMul(x,y)
104 }
105
106 class PointSumMulDiv(val x:Int, val y:Int) extends tPointSum with tPointMul with tPointDiv {
107   override type p = PointSumMulDiv
108   override def of(x:Int,y:Int)=
109     new PointSumMulDiv(x,y)
110 }
111
112 class PointSubMulDiv(val x:Int, val y:Int) extends tPointSub with tPointMul with tPointDiv {
113   override type p = PointSubMulDiv
114   override def of(x:Int,y:Int)=
115     new PointSubMulDiv(x,y)
116 }
117
118 class PointSumSubMulDiv(val x:Int, val y:Int) extends tPointSum with tPointSub with tPointMul with tPointDiv {
119   override type p = PointSumSubMulDiv
120   override def of(x:Int,y:Int)=
121     new PointSumSubMulDiv(x,y)
122 }

```

#### C.1.4 42

The main differences with respect to 42<sub>μ</sub> as presented in the paper are: we use `Resource<><` to declare a trait, all trait names start with \$, traits are referred to with a ‘()’. Method bodies can be simple expressions, without the need for a ‘return’. Note that in full 42 there is no primitive type `int`, here we use `Num` instead, which is an arbitrary precision rational.

```

1 $p: Resource <>< { //4
2   method Num x()
3   method Num y()
4   class method This of(Num x,Num y)
5 }
6
7 $pointSum: Resource <>< Use[$p()] <>< { //3*4
8   method This sum(This that)
9   This.of(x: this.x()+that.x(), y: this.y()+that.y())
10 }
11
12 $pointSub: Resource <>< Use[$p()] <>< {
13   method This sub(This that)
14   This.of(x: this.x()-that.x(), y: this.y()-that.y())
15 }
16
17 $pointMul: Resource <>< Use[$p()] <>< {
18   method This mul(This that)
19   This.of(x: this.x()*that.x(), y: this.y()*that.y())
20 }
21
22 $pointDiv: Resource <>< Use[$p()] <>< {
23   method This div(This that)
24   This.of(x: this.x()/that.x(), y: this.y()/that.y())
25 }
26
27 Point: Use[$p()] <>< { //1*16
28
29 PointSum: Use[$pointSum()] <>< {
30
31 PointSub: Use[$pointSub()] <>< {
32
33 PointMul: Use[$pointMul()] <>< {
34
35 PointDiv: Use[$pointDiv()] <>< {
36
37 PointSumSub: Use[$pointSum();$pointSub()] <>< {
38
39 PointSumMul: Use[$pointSum();$pointMul()] <>< {
40
41 PointSumDiv: Use[$pointSum();$pointDiv()] <>< {
42
43 PointSubMul: Use[$pointSub();$pointMul()] <>< {
44
45 PointSubDiv: Use[$pointSub();$pointDiv()] <>< {

```

## Separating Use and Reuse to Improve Both

```
46
47 PointMulDiv: Use[$pointMul();$pointDiv()] <>< {}
48
49 PointSumSubDiv: Use[$pointSum();$pointSub();$pointDiv()] <>< {}
50
51 PointSumSubMul: Use[$pointSum();$pointSub();$pointMul()] <>< {}
52
53 PointSumMulDiv: Use[$pointSum();$pointMul();$pointDiv()] <>< {}
54
55 PointSubMulDiv: Use[$pointSub();$pointMul();$pointDiv()] <>< {}
56
57 PointSumSubMulDiv: Use[$pointSum();$pointSub();$pointMul();$pointDiv()] <>< {}
```

## C.2 FCPoint

### C.2.1 Java7

```
1 enum Flavor {
2     NONE, SOUR, SWEET, SALTY, SPLICY;
3 }
4
5 class Color {
6     final int r;
7     final int g;
8     final int b;
9
10    public Color(int r, int g, int b) {
11        this.r = r;
12        this.g = g;
13        this.b = b;
14    }
15
16    public Color mix(Color that) {
17        return new Color((this.r + that.r) / 2, (this.g + that.g) / 2, (this.b + that.b) / 2);
18    }
19 }
20
21 class Point { //10 lines
22     final int x;
23     final int y;
24
25     public Point(int x, int y) {
26         this.x = x;
27         this.y = y;
28     }
29
30     public Point withX(int that) {
31         return new Point(that, this.y);
32     }
33
34     public Point withY(int that) {
35         return new Point(this.x, that);
36     }
37 }
38
39 class PointSum extends Point { //9 lines
40     public PointSum(int x, int y) {
41         super(x, y);
42     }
43
44     public PointSum withX(int that) {
45         return new PointSum(that, this.y);
46     }
47
48     public PointSum withY(int that) {
49         return new PointSum(this.x, that);
50     }
51
52     public PointSum sum(Point that) {
53         return this.withX(this.x + that.x).withY(this.y + that.y);
54     }
55 }
56
57 class CPoint extends PointSum { //13 lines
58     final Color color;
59
60     public CPoint(int x, int y, Color color) {
61         super(x, y);
62         this.color = color;
63     }
64
65     public CPoint withX(int that) {
66         return new CPoint(that, this.y, this.color);
67     }
68
69     public CPoint withY(int that) {
```



```

70 return new CPoint(this.x, that, this.color);
71 }
72
73 public CPoint withColor(Color that) {
74     return new CPoint(this.x, this.y, that);
75 }
76
77 public CPoint merge(CPoint that) {
78     return this.withColor(this.color.mix(that.color));
79 }
80 }
81
82 class FCPPoint extends CPoint { //15 lines
83     final Flavor flavor;
84
85     public FCPPoint(int x, int y, Color color, Flavor flavor) {
86         super(x, y, color);
87         this.flavor = flavor;
88     }
89
90     public FCPPoint withX(int that) {
91         return new FCPPoint(that, this.y, this.color, this.flavor);
92     }
93
94     public FCPPoint withY(int that) {
95         return new FCPPoint(this.x, that, this.color, this.flavor);
96     }
97
98     public FCPPoint withColor(Color that) {
99         return new FCPPoint(this.x, this.y, that, this.flavor);
100    }
101
102    public FCPPoint withFlavor(Flavor that) {
103        return new FCPPoint(this.x, this.y, this.color, that);
104    }
105
106    public FCPPoint merge(FCPPoint that) {
107        return this.withColor(that.color).withFlavor(that.flavor);
108    }
109 }

```

## C.2.2 Scala

```

1 trait tPointState { //8 lines
2     type p <: tPointState
3     def x: Int
4     def y: Int
5     def withX(that: Int): p
6     def withY(that: Int): p
7     def of(x: Int, y: Int): p
8     def merge(that: p): p
9 }
10
11 trait tPointSum extends tPointState { //3
12     def sum(that: p) =
13         this.merge(that).withX(this.x+that.x).withY(this.y+that.y)
14 }
15
16 trait tColored { //6*2
17     type p <: tColored
18     def color: Int
19     def withColor(that: Int): p
20     def merge(that: p) =
21         this.withColor(this.color+that.color)
22 }
23
24 trait tFlavored {
25     type p <: tFlavored
26     def flavor: Int
27     def withFlavor(that: Int): p
28     def merge(that: p) =
29         this.withFlavor(that.flavor)
30 }
31 //glue code from now on
32 class CPoint(val x: Int, val y: Int, val color: Int) extends tPointSum with tColored { //12 lines
33     override type p = CPoint
34     def of(x: Int, y: Int) =
35         this.of(x, y, 0)
36     def of(x: Int, y: Int, color: Int) =
37         new CPoint(x, y, color)
38     def withX(that: Int) =
39         of(that, y, color)
40     def withY(that: Int) =
41         of(x, that, color)
42     def withColor(that: Int) =
43         of(x, y, that)
44 }
45 //18 lines

```

## Separating Use and Reuse to Improve Both

```
46 class FCPoint(val x:Int, val y:Int, val color:Int, val flavor:Int) extends tPointSum with tColored with tFlavored{
47   override type p = FCPoint
48   def of(x:Int,y:Int)=
49     this.of(x,y,0,0)
50   def of(x:Int,y:Int,color:Int,flavor:Int)=
51     new FCPoint(x,y,color,flavor)
52   def withX(that:Int)=
53     of(that,y,color,flavor)
54   def withY(that:Int)=
55     of(x,that,color,flavor)
56   def withColor(that:Int)=
57     of(x,y,that,flavor)
58   def withFlavor(that:Int)=
59     of(x,y,color,that)
60   def superMergeFlavoured(that:FCPoint)=
61     super[tFlavored].merge(that)
62   override def merge(that:FCPoint)=
63     super[tColored].merge(that).superMergeFlavoured(that)
64 }
```

### C.2.3 42

```
1 Flavor: Enumeration"sour, sweet, salty, spicy"
2
3 Color:Data<><{Num r Num g Num b
4   method This mix(This that)
5     Color( r:(this.r()+that.r())/2Num, g:(this.g()+that.g())/2Num, b:(this.b()+that.b())/2Num)
6   }
7
8 $p: Resource <>< { // 7 lines
9   method Num x()
10  method Num y() //getters
11  method This withX(Num that)
12  method This withY(Num that)//withers
13  class method This of(Num x,Num y)
14  method This merge(This that) //new method merge!
15 }
16
17 $pointSum: Resource <>< Use[$p()] <>< { // 3 lines
18   method This sum(This that)
19     this.merge(that).withX(this.x()+that.x()).withY(this.y()+that.y())
20 }
21
22 $colored: Resource <>< { // 5 +2 lines
23   method Color color()
24   method This withColor(Color that)
25   method This merge(This that) //how to merge colors
26   this.withColor(this.color().mix(that.color()))
27 }
28
29 $flavored: Resource <>< {
30   method Flavor flavor() //very similar to colored
31   method This withFlavor(Flavor that)
32   method This merge(This that) //how to merge flavors
33   this.withFlavor(that.flavor())//inherits "that" flavor
34 }
35
36 CPoint: Use[$pointSum();$colored()] <>< { // 4 lines
37   class method This of(Num x,Num y)
38     This.of(x:x,y:y,color:Color(r:100Num,g:0Num,b:0Num))
39   class method This of(Num x, Num y,Color color)
40 }
41
42 FCPoint: Use[//9 lines
43   Refactor2.toAbstract(selector:"merge(that)", into:"_1merge(that)")<><$colored();
44   Refactor2.toAbstract(selector:"merge(that)", into:"_2merge(that)")<><$flavored();
45   $pointSum()] <>< {
46   class method This of(Num x,Num y)
47     This.of(x:x,y:y,color:Color(r:100Num,g:0Num,b:0Num),flavor:Flavor.sour())
48   class method This of(Num x, Num y,Color color,Flavor flavor)
49     method This merge(This that)
50     this._1merge(that)._2merge(that)
51 }
```

## C.3 Expression problem

### C.3.1 Original Scala

Here is the original code of Scala solving the expression problem. As you can see, sometimes multiple operations/cases are declared together in the same trait.

```
1 trait Base {
```

```

2  type exp <: Exp
3
4  trait Exp {
5    def eval: Int
6  }
7
8  class Num(val value: Int) extends Exp {
9    def eval: Int = value
10 }
11
12 type BaseNum = Num
13 }
14
15 trait BasePlus extends Base {
16 class Plus(val left: exp, val right: exp) extends Exp {
17   def eval: Int = left.eval + right.eval
18 }
19
20 type BasePlus = Plus
21 }
22
23 trait BaseNeg extends Base {
24 class Neg(val term: exp) extends Exp {
25   def eval = -term.eval
26 }
27
28 type BaseNeg = Neg
29 }
30 trait BasePlusNeg extends BasePlus with BaseNeg
31 trait Show extends Base {
32   type exp <: Exp
33
34   trait Exp extends super.Exp {
35     def show: String
36   }
37
38   trait NumBehavior extends Exp {
39     self: BaseNum =>
40     override def show: String = value.toString
41   }
42
43   final class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
44 }
45
46 trait ShowPlusNeg extends BasePlusNeg with Show {
47   trait PlusBehavior {
48     self: BasePlus =>
49     def show = left.show + "+" + right.show;
50   }
51
52   final class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
53
54   trait NegBehavior {
55     self: BaseNeg =>
56     def show = "-" + term.show + ";";
57   }
58
59   class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
60 }
61
62 trait DblePlusNeg extends BasePlusNeg {
63   type exp <: Exp
64
65   trait Exp extends super.Exp {
66     def dble: exp
67   }
68
69   def Num(v: Int): exp
70
71   def Plus(l: exp, r: exp): exp
72
73   def Neg(t: exp): exp
74
75   trait NumBehavior {
76     self: BaseNum =>
77     def dble = Num(value * 2)
78   }
79
80   final class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
81
82   trait PlusBehavior {
83     self: BasePlus =>
84     def dble = Plus(left.dble, right.dble)
85   }
86
87   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
88
89   trait NegBehavior {
90     self: BaseNeg =>
91     def dble = Neg(term.dble)
92 }

```

## Separating Use and Reuse to Improve Both

```
93
94 class Neg(t: exp) extends super.Neg(t) with NegBehavior with Exp
95 }
96 /-- 52 lines up to here, not counting new lines and '*/
97 trait Equals extends Base {
98   type exp <: Exp;
99
100  trait Exp extends super.Exp {
101    def eql(other: exp): Boolean;
102
103    def isNum(v: Int): Boolean = false;
104  }
105
106  trait NumBehavior extends Exp {
107    self: BaseNum =>
108    def eql(other: exp): Boolean = other.isNum(value);
109
110    override def isNum(v: Int) = v == value;
111  }
112
113  final class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
114 }
115
116 trait EqualsPlusNeg extends BasePlusNeg with Equals {
117   type exp <: Exp;
118
119   trait Exp extends super[BasePlusNeg].Exp
120     with super[Equals].Exp {
121     def isPlus(l: exp, r: exp): Boolean = false;
122
123     def isNeg(t: exp): Boolean = false;
124   }
125
126   final class Num(v: Int) extends BaseNum(v)
127     with NumBehavior // effectively super[Equals].NumBehavior
128     with Exp
129
130   trait PlusBehavior extends Exp {
131     self: BasePlus =>
132     def eql(other: exp): Boolean = other.isPlus(left, right);
133
134     override def isPlus(l: exp, r: exp) = (left eql l) && (right eql r)
135   }
136
137   final class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
138
139   trait NegBehavior extends Exp {
140     self: BaseNeg =>
141     def eql(other: exp): Boolean = other.isNeg(term);
142
143     override def isNeg(t: exp) = term eql t
144   }
145
146   final class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
147 }
148
149 trait EqualsShowPlusNeg extends EqualsPlusNeg with ShowPlusNeg {
150   type exp <: Exp
151
152   trait Exp extends super[EqualsPlusNeg].Exp
153     with super[ShowPlusNeg].Exp
154
155   trait NumBehavior extends super[EqualsPlusNeg].NumBehavior with super[ShowPlusNeg].NumBehavior {
156     self: BaseNum =>
157   }
158
159   class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
160
161   trait PlusBehavior extends super[EqualsPlusNeg].PlusBehavior with super[ShowPlusNeg].PlusBehavior {
162     self: BasePlus =>
163   }
164
165   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
166
167   trait NegBehavior extends super[EqualsPlusNeg].NegBehavior with super[ShowPlusNeg].NegBehavior {
168     self: BaseNeg =>
169   }
170
171   class Neg(term: exp) extends BaseNeg(term) with NegBehavior with Exp
172 }
173 /-- 40 lines for equals
```

### C.3.2 Isolated Scala

The following code is fully modularized: every trait defines exactly one operation for each datavariant.

```

1 trait Base {
2   type exp <: Exp
3
4   trait Exp { }
5 }
6
7 trait BaseNum extends Base {
8   class Num(val value: Int) extends Exp { }
9
10  type BaseNum = Num
11 }
12
13 trait BasePlus extends Base {
14   class Plus(val left: exp, val right: exp) extends Exp { }
15
16   type BasePlus = Plus
17 }
18
19 trait BaseNeg extends Base {
20   class Neg(val term: exp) extends Exp { }
21
22   type BaseNeg = Neg
23 }
24 //-----EVAL
25 trait Eval extends Base {
26   type exp <: Exp
27
28   trait Exp extends super.Exp {
29     def eval: Int
30   }
31 }
32
33 //-----EVALNUM
34 trait EvalNum extends BaseNum with Eval {
35   trait NumBehavior {
36     self: BaseNum =>
37     def eval: Int = value
38   }
39
40   class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
41 }
42
43 //-----EVALPLUS
44 trait EvalPlus extends BasePlus with Eval {
45   trait PlusBehavior {
46     self: BasePlus =>
47     def eval = left.eval + right.eval;
48   }
49
50   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
51 }
52 //-----EVALNEG
53 trait EvalNeg extends BaseNeg with Eval {
54   trait NegBehavior {
55     self: BaseNeg =>
56     def eval = - term.eval;
57   }
58
59   class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
60 }
61 //-----SHOW
62 trait Show extends Base {
63   type exp <: Exp
64
65   trait Exp extends super.Exp {
66     def show: String
67   }
68 }
69 //-----SHOWNUM
70 trait ShowNum extends BaseNum with Show {
71   trait NumBehavior {
72     self: BaseNum =>
73     def show: String = value.toString
74   }
75
76   class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
77 }
78
79 //-----SHOWPLUS
80 trait ShowPlus extends BasePlus with Show {
81   trait PlusBehavior {
82     self: BasePlus =>
83     def show = left.show + "+" + right.show;
84   }
85
86   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
87 }
88 //-----SHOWNEG
89 trait ShowNeg extends BaseNeg with Show {
90   trait NegBehavior {
91     self: BaseNeg =>

```

## Separating Use and Reuse to Improve Both

```
92 def show = "-" + term.show + ";
93 }
94
95 class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
96 }
97 //-----DBLE
98 trait Dble extends Base {
99   type exp <: Exp
100
101   trait Exp extends super.Exp {
102     def dble: exp
103   }
104 }
105 //-----DBLENUM
106 trait DbleNum extends BaseNum with Dble {
107   type exp <: Exp
108
109   trait Exp extends super[BaseNum].Exp with super[Dble].Exp
110
111   trait NumBehavior {
112     self: BaseNum =>
113     def dble = Num(value * 2)
114   }
115
116   def Num(v: Int): exp
117
118   class Num(v: Int) extends super.Num(v) with NumBehavior
119 }
120 //-----DBLEPLUS
121 trait DblePlus extends BasePlus with Dble {
122   type exp <: Exp
123
124   trait Exp extends super[BasePlus].Exp with super[Dble].Exp
125
126   trait PlusBehavior {
127     self: BasePlus =>
128     def dble = Plus(left.dble, right.dble)
129   }
130
131   def Plus(l: exp, r: exp): exp
132
133   class Plus(l: exp, r: exp) extends super.Plus(l, r) with PlusBehavior with Exp
134 }
135 //-----DBLENEG
136 trait DbleNeg extends BaseNeg with Dble {
137   type exp <: Exp
138
139   trait Exp extends super[BaseNeg].Exp with super[Dble].Exp
140
141   trait NegBehavior {
142     self: BaseNeg =>
143     def dble = Neg(term.dble)
144   }
145
146   def Neg(t: exp): exp
147
148   class Neg(t: exp) extends super.Neg(t) with NegBehavior with Exp
149 }//78 lines up to here, not counting new lines and '}'
150
151
152 //glue code: 27 lines
153 object All0 extends
154   Eval with Show with Dble with
155   EvalNum with EvalPlus with EvalNeg with
156   ShowNum with ShowPlus with ShowNeg with
157   DbleNum with DblePlus with DbleNeg {
158   override type exp = Exp
159
160   trait Exp extends
161     super[Eval].Exp with super[Show].Exp with super[Dble].Exp with
162     super[DbleNum].Exp with super[DblePlus].Exp with super[DbleNeg].Exp
163
164   trait NumBehavior extends
165     super[EvalNum].NumBehavior with super[ShowNum].NumBehavior with
166     super[DbleNum].NumBehavior {
167     self: BaseNum =>
168   }
169
170   class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
171
172   trait PlusBehavior extends
173     super[EvalPlus].PlusBehavior with super[ShowPlus].PlusBehavior with
174     super[DblePlus].PlusBehavior {
175     self: BasePlus =>
176   }
177
178   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
179
180   trait NegBehavior extends
181     super[EvalNeg].NegBehavior with super[ShowNeg].NegBehavior with
182     super[DbleNeg].NegBehavior {
```

```

183   self: BaseNeg =>
184   }
185
186   class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
187
188   def Num(v: Int) = new Num(v)
189
190   def Plus(l: exp, r: exp) = new Plus(l,r)
191
192   def Neg(t: exp) = new Neg(t)
193 }
194 //-----EQUALS
195 trait Equals extends Base {
196   type exp <: Exp;
197
198   trait Exp extends super.Exp {
199     def eql(other: exp): Boolean;
200   }
201 }
202 //-----EQUALSNUM
203 trait EqualsNum extends BaseNum with Equals {
204   type exp <: Exp;
205
206   trait Exp extends super.Exp {
207     def isNum(v: Int): Boolean = false;
208   }
209
210   trait NumBehavior extends Exp {
211     self: BaseNum =>
212     def eql(other: exp): Boolean = other.isNum(value);
213
214     override def isNum(v: Int) = v == value;
215   }
216
217   class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
218 }
219 //-----EQUALSPPLUS
220 trait EqualsPlus extends BasePlus with Equals {
221   type exp <: Exp;
222
223   trait Exp extends super[BasePlus].Exp with super[Equals].Exp {
224     def isPlus(l: exp, r: exp): Boolean = false;
225   }
226
227   trait PlusBehavior extends Exp {
228     self: BasePlus =>
229     def eql(other: exp): Boolean = other.isPlus(left, right);
230
231     override def isPlus(l: exp, r: exp) = (left eql l) && (right eql r)
232   }
233
234   class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
235 }
236 //-----EQUALSNEG
237 trait EqualsNeg extends BaseNeg with Equals {
238   type exp <: Exp;
239
240   trait Exp extends super[BaseNeg].Exp with super[Equals].Exp {
241     def isNeg(t: exp): Boolean = false;
242   }
243
244   trait NegBehavior extends Exp {
245     self: BaseNeg =>
246     def eql(other: exp): Boolean = other.isNeg(term);
247
248     override def isNeg(t: exp) = term eql t
249   }
250
251   class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
252 }
253 //31 lines for equals, 29 glue
254 object All extends
255   Eval with Show with Dble with Equals with
256   EvalNum with EvalPlus with EvalNeg with
257   ShowNum with ShowPlus with ShowNeg with
258   DbleNum with DblePlus with DbleNeg with
259   EqualsNum with EqualsPlus with EqualsNeg
260 {
261   override type exp = Exp
262
263   trait Exp extends
264     super[Eval].Exp with super[Show].Exp with super[Dble].Exp with super[Equals].Exp with
265     super[DbleNum].Exp with super[DblePlus].Exp with super[DbleNeg].Exp with
266     super[EqualsNum].Exp with super[EqualsPlus].Exp with super[EqualsNeg].Exp
267
268   trait NumBehavior extends
269     super[EvalNum].NumBehavior with super[ShowNum].NumBehavior with
270     super[DbleNum].NumBehavior with super[EqualsNum].NumBehavior {
271     self: BaseNum =>
272   }
273

```

## Separating Use and Reuse to Improve Both

```
274 class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
275
276 trait PlusBehavior extends
277   super[EvalPlus].PlusBehavior with super>ShowPlus'.PlusBehavior with
278   super[DblePlus].PlusBehavior with super[EqualsPlus].PlusBehavior {
279   self: BasePlus =>
280 }
281
282 class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
283
284 trait NegBehavior extends
285   super[EvalNeg].NegBehavior with super>ShowNeg'.NegBehavior with
286   super[DbleNeg].NegBehavior with super[EqualsNeg].NegBehavior {
287   self: BaseNeg =>
288 }
289
290 class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
291
292 def Num(v: Int) = new Num(v)
293
294 def Plus(l: exp, r: exp) = new Plus(l,r)
295
296 def Neg(t: exp) = new Neg(t)
297 }
```

### C.3.3 42

The classes `RenNum`, `RenPlus`, and `RenNeg` can be understood as declarations of short cuts in order to make the full 42 language more in line with the more compact syntax of  $42_{\mu}$ . In 42, these short cuts can be very expressive but here we use them only to better align the full 42 and  $42_{\mu}$  languages.

```
1 RenNum:{class method Library<<< (Library that) exception Guard
2 Refactor2.rename(path:"T" into:"Num")<<<that
3 }
4
5 RenPlus:{class method Library<<< (Library that) exception Guard
6 Refactor2.rename(path:"T" into:"Plus")<<<that
7 }
8
9 RenNeg:{class method Library<<< (Library that) exception Guard
10 Refactor2.rename(path:"T" into:"Neg")<<<that
11 }
12 //above, header not counted
13
14 $exp:Resource<<<{ //3 lines
15 Exp:{interface}
16
17 T:{implements Exp}
18 }
19
20 $num:Resource<<<Use[RenNum]<<<$exp())<<<{ //4 lines
21 Num:{
22   method Size value()
23   class method Num of(Size value)
24 }
25 }
26
27 $plus:Resource<<<Use[RenPlus]<<<$exp())<<<{ //6 lines
28 Exp:{interface}
29
30 Plus:{
31   method Exp left()
32   method Exp right()
33   class method Plus of(Exp left, Exp right)
34 }
35 }
36
37 $neg:Resource<<<Use[RenNeg]<<<$exp())<<<{ //5 lines
38 Exp:{interface}
39
40 Neg:{
41   method Exp term()
42   class method Neg of(Exp term)
43 }
44 }
45
46 $eval:Resource<<< { //4 lines
47 Exp:{interface
48   method Size eval()
49 }
50
51 T:{implements Exp}
52 }
53 }
```



```

54 $evalNum:Resource<><Use[$num(); RenNum<<$eval()]>>< { // 2*3 lines
55 Num:{ method Size eval() this.value() }
56 }
57
58 $evalPlus:Resource<><Use[$plus(); RenPlus<<$eval()]>>< {
59 Plus:{ method Size eval() this.left().eval()+this.right().eval() }
60 }
61
62 $evalNeg:Resource<><Use[$neg(); RenNeg<<$eval()]>>< {
63 Neg:{ method Size eval() Size"-1" * this.term().eval() }
64 }
65
66 $show:Resource<><Use[$exp()]>><{ //4 lines
67 Exp:{interface
68 method S show()
69 }
70
71 T:{implements Exp}
72 }
73
74 $showNum:Resource<><Use[$num(); RenNum<<$show()]>><{ //2*3 lines
75 Num:{ method S show() this.value().toS() }
76 }
77
78 $showPlus:Resource<><Use[$plus(); RenPlus<<$show()]>><{
79 Plus:{ method S show() this.left().show()+S" + "+this.right().show() }
80 }
81
82 $showNeg:Resource<><Use[$neg(); RenNeg<<$show()]>><{
83 Neg:{ method S show() S"-("+this.term().show()+S")" }
84 }
85 //-----
86 $double:Resource<><{ //4 lines
87 Exp:{interface
88 method Exp double()
89 }
90
91 T:{implements Exp}
92 }
93
94 $doubleNum:Resource<><Use[$num(); RenNum<<$double()]>><{ //2*3 lines
95 Exp:{interface};//not counted, needed in the full language only to guide scope resolution in the desugaring
96
97 Num:{ method Exp double() Num.of(value: this.value()*2Size) }
98 }
99
100 $doublePlus:Resource<><Use[$plus(); RenPlus<<$double()]>><{
101 Exp:{interface}
102
103 Plus:{ method Exp double() Plus.of(left: this.left().double(), right: this.right().double()) }
104 }
105
106 $doubleNeg:Resource<><Use[$neg(); RenNeg<<$double()]>><{
107 Exp:{interface}
108
109 Neg:{ method Exp double() Neg.of(term: this.term().double()) }
110 }
111 //-----
112 $equals:Resource<><Use[$eval()]>><{ //6 lines
113 Exp:{interface
114 method Bool eql(Exp that)
115 method Bool equalToT(T that)
116 }
117
118 T:{implements Exp
119 method eql(that) that.equalToT(this)
120 }
121 }
122
123 $equalsNum:Resource<><Use[$num(); //5*3 lines
124 Refactor2.Method[rename:"equalToT(that)" of:"Exp" into:"equalToNum(that)"]
125 <<<RenNum<<$equals()]>><{
126 Num:{
127 method Bool equalToNum(Num that) this.value()==that.value()
128 }
129 }
130
131 $equalsPlus:Resource<><Use[$plus();
132 Refactor2.Method[rename:"equalToT(that)" of:"Exp" into:"equalToPlus(that)"]
133 <<<RenPlus<<$equals()]>><{
134 Plus:{
135 method Bool equalToPlus(Plus that)
136 this.left().eql(that.left()) & this.right().eql(that.right())
137 }
138 }
139
140 $equalsNeg:Resource<><Use[$neg();
141 Refactor2.Method[rename:"equalToT(that)" of:"Exp" into:"equalToNeg(that)"]
142 <<<RenNeg<<$equals()]>><{
143 Neg:{
144 method Bool equalToNeg(Neg that) this.term().eql(that.term())

```

## Separating Use and Reuse to Improve Both

```
145 }
146 }
147
148 //16+6 lines glue code
149 $equalsAll:Resource<><<Use[$equalsNeg();$equalsPlus();$equalsNum()]>><<{
150 Exp:{interface
151   method Bool eql(Exp that)
152   method Bool equalToNum(Num that)
153   method Bool equalToPlus(Plus that)
154   method Bool equalToNeg(Neg that)
155   method Size eval()
156 }
157
158 Num:{implements Exp
159   method equalToPlus(that) Bool.false()
160   method equalToNeg(that) Bool.false()
161 }
162
163 Plus:{implements Exp
164   method equalToNum(that) Bool.false()
165   method equalToNeg(that) Bool.false()
166 }
167
168 Neg:{implements Exp
169   method equalToNum(that) Bool.false()
170   method equalToPlus(that) Bool.false()
171 }
172 }
173
174 $evalAll:Resource<><<Use[$evalPlus();$evalNeg()]>><<$evalNum()
175
176 $showAll:Resource<><<Use[$showNum();$showPlus()]>><<$showNeg()
177
178 $doubleAll:Resource<><<Use[$doubleNum();$doublePlus()]>><<$doubleNeg()
179
180 ESDAll0:Use[Use[$evalPlus();$evalNeg()]>><<$evalNum();
181   Use[$showNum();$showPlus()]>><<$showNeg();
182   $doubleNum();$doublePlus()]>><<$doubleNeg()
183
184
185 ESDAll:Use[$evalAll();$showAll()]>><<$doubleAll()
186
187 ESDEAll:Use[$evalAll();$showAll();$doubleAll()]>><<$equalsAll()
188
189 $equals2:Resource<><<{ //7 lines
190 Exp:{interface method Bool equals(Exp that)}
191 T:{implements Exp
192   method Bool exactEquals(T that)
193   method equals(that){
194     with that (on T return this.exactEquals(that) )
195     return Bool.false()
196   }
197 }
198 }
199
200 $equalsNum2:Resource<><<Use[$num();RenNum<><<$equals2()]>><<{ //2*3 lines
201   Num:{ method Bool exactEquals(Num that) this.value().equals(that.value()) }
202 }
203
204 $equalsPlus2:Resource<><<Use[$plus();RenPlus<><<$equals2()]>><<{
205   Plus:{ method Bool exactEquals(Plus that) this.left().equals(that.left()) & this.right().equals(that.right()) }
206 }
207
208 $equalsNeg2:Resource<><<Use[$neg();RenNeg<><<$equals2()]>><<{
209   Neg:{method Bool exactEquals(Neg that) this.term().equals(that.term()) }
210 }
211
212 $equalsAll2:Resource<><<Use[$equalsNum2();$equalsPlus2()]>><<$equalsNeg2()
213
214 ESDEAll2:Use[$evalAll();$showAll();$doubleAll()]>><<$equalsAll2()
```