Separating Use and Reuse to Improve Both

Hrshikesh Arora\textsuperscript{a}, Marco Servetto\textsuperscript{a}, and Bruno C. d. S. Oliveira\textsuperscript{b}

\textsuperscript{a} Victoria University of Wellington
\textsuperscript{b} 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 \textit{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 \cite{6}, Package Templates \cite{26} and DeepFJig \cite{16}.

Approach: In this paper, we design 42\textsubscript{µ}, a new language, where we improve use and reuse and support the \textit{This} type and family polymorphism by distinguishing code designed for use from code designed for reuse. In this way 42\textsubscript{µ} synthesise the 3 approaches above, and improves them with \textit{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\textsubscript{µ}'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
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.

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

Now, lets add a class `B`

```java
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`.  

```java
```
Suppose Java code-reuse (the `extends` keyword) did not introduce subtyping\(^1\): 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

---

\(^1\) 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\textmu{} 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\textmu{} 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 subtyping 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\textmu{}, 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\textmu{}: Separating Use and Reuse

### 2.1 Classes in 42\textmu{}: a mechanism for code use

Consider the example of Section 1 rewritten in 42\textmu{}, introducing classes \texttt{Utils} and \texttt{A}:

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

Classes in 42\textmu{} use a different declaration style compared to Java: there is no \texttt{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 \texttt{A}, the name of the class is \texttt{A} and the code literal associated with the class is \texttt{‘{ method int ma()\{return Utils.m(this);\}’}
and it contains the method \texttt{ma()}. In the 42\textmu code above, there is no way to add a class \texttt{B} reusing the code of \texttt{A}; class \texttt{A} (uppercase) is designed for code use and not reuse. Indeed, a noticeable difference with Java is the absence of the \texttt{extends} keyword. 42\textmu 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 \texttt{B} is not possible. This may seem like a severe restriction, but 42\textmu has a different mechanism for code-reuse that is more appropriate when code-reuse is intended.

\subsection*{2.2 Traits in 42\textmu: a mechanism for code reuse}

Traits in 42\textmu 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).

\begin{verbatim}
Utils= { static method int m(A a){return ...} }
ta={ method int ma(){return Utils.m(this);}} //type error
A=Use ta
\end{verbatim}

Here \texttt{ta} is a trait intended to replace the original class \texttt{A} so that the code of the method \texttt{ma} can be reused. Then the class \texttt{A} is created by reusing the code from the trait \texttt{ta}, introduced by the keyword \texttt{Use}. Note that \texttt{Use} expressions cannot contain class names: only trait names are allowed. \textit{Referring to a trait is the only way to induce code reuse.}

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

The following second attempt would not work either:

\begin{verbatim}
Utils= { static method int m(ta a){return ...} } //syntax error
A=Use ta
\end{verbatim}

\texttt{ta} is not a type in the first place, since it is a (lowercase) trait name. Indeed, trait names can only be used in \texttt{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\textmu: traits can be reused in multiple places, and their code can be seen under multiple types. In 42\textmu, \textit{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 \texttt{A} and \texttt{B}:

\begin{verbatim}
IA={ interface method int ma() } //interface with abstract method
Utils= { static method int m(IA a){return ...} } 
ta={ implements IA //This line is the core of the solution
     method int ma(){return Utils.m(this);}}
A=Use ta
\end{verbatim}

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

\begin{verbatim}
B=Use ta, { method int mb(){return this.ma();} }
\end{verbatim}
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:

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

In 42ₜ, methods are directly accessible from ta, exactly as in the Java equivalent

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

where method ma is imported from A. This concept is natural for a Java programmer, but was not supported in previous work [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.

```
A={implements IA method int ma(){return Utils.m(this);}}
B={implements IA
    method int ma(){return Utils.m(this);}
    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 42ₜ, 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.
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:

```java
class Set { // usual hashmap implementation
    private Elem[] hashMap;
    void put(Elem e){...}
    boolean isIn(Elem e){...}
}
class Bag extends Set { // for each element in the hash map,
    private int[] countMap; // keep track of how many occurrences are in the bag
    @Override void put(Elem e){...}
    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! Indeed, the following is allowed:

```java
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:

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

This looks unnatural, since `Set` would extend `AbstractSetOrBag` without adding anything, and we would be surprised to find a use of the type `AbstractSetOrBag`. Worst, if we are to constantly apply this 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

---

2 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 $42\_\mu$

Instead, in $42\_\mu$, if we were to originally declare

```java
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 undocumented behaviour nearly all the time! If the designer of the `Set` class wishes to make it reusable, they can do it explicitly by providing a set trait:

```java
set={/*set implementation*/}
Set=Use set
```

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

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

or

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

Notice how, thanks to flattening, the resulting code for `Bag` is identical in both versions and, as shown in Section 2, there is no trace of trait `bag` 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.

---

3 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.
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 $y$.

```java
// idealized state of the art trait language, not 42
public class PointSum {
    int x();
    int y();

    static This of(int x, int y) // factory method
    static This sum(This that){
        return This.of(this.x()+that.x(),this.y()+that.y()); // self instantiation
    }

    void mul(This that){
        return This.of(this.x()*that.x(),this.y()*that.y());
    }
}
```

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 \( \text{of} \). This approach is verbose but very powerful, as illustrated by Wang et al. [40].

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

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 \( \text{This} \). The class will then implement the methods returning \( \text{This} \) 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, there is no need to write down constructors and fields. In fact, in 42, 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 \( \text{This} \) and with parameters \( T_1 \ x_1, \ldots, T_n \ x_n \) is coherent if all the other abstract methods can be seen as abstract state operations over one of \( x_1, \ldots, x_n \). That is:
  - A method \( T_i \ x_i() \) is interpreted as an abstract state method: a getter for \( x_i \).
  - A method \( \text{void} \ x_i(T_i \ that) \) 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 \( \text{that} \). 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, \( \text{pointSum} \) and \( \text{pointMul} \) can indeed be directly composed. This works because the resulting class is coherent.

```
PointAlgebra = Use \text{pointSum}, \text{pointMul}  // no glue code needed
```

**Improved solution:** So far the current solution still repeats the abstract methods \( x \), \( y \) and \( \text{of} \). Moreover, in addition to \( \text{sum} \) and \( \text{mul} \) 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 \( p \):

```
// one
p = { method int x() method int y() }
// two
static method This of(int x, int y)
// three
pointSum = Use p, {
  method This sum(This that){
    return This.of(this.x()+that.x(),this.y()+that.y());
  }
// four
pointMul = Use p, {
  method This mul(This that){
    return This.of(this.x()*that.x(),this.y()*that.y());
  }
// five
pointDiv = ...
// six
PointAlgebra = Use \text{pointSum}, \text{pointMul}, \text{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 \( \text{Sum} \), \( \text{Subtraction} \), \( \text{Multiplication} \) and \( \text{Division} \). 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). 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

---

\(^4\) 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.
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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of code</th>
<th>N. of members</th>
<th>N. classes/traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java7</td>
<td>$115 = 6 + 5 \times 4 + 7 \times 6 + 9 \times 4 + 11$</td>
<td>50</td>
<td>16</td>
</tr>
<tr>
<td>Classless Java</td>
<td>$82 = 3 + 3 \times 4 + 5 \times 6 + 7 \times 4 + 9$</td>
<td>34</td>
<td>16</td>
</tr>
<tr>
<td>Scala</td>
<td>$81 = 5 + 3 \times 4 + 4 \times 16$</td>
<td>40</td>
<td>$21 = 16 + 4 + 1$</td>
</tr>
<tr>
<td>$42_\mu$</td>
<td>$32 = 4 + 3 \times 4 + 1 \times 16$</td>
<td>7</td>
<td>$21 = 16 + 4 + 1$</td>
</tr>
</tbody>
</table>

We observed that in Java7 we had to duplicate $^5$ 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) $+ 3 \times 4$ (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:

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

$^5$ 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 \texttt{static method This of(int x, int y)}.

A solution is to provide a richer factory:

```java
CPoint = Use pointSum, colored,
  static method This of(int x, int y) [return This.of(x, y, Color.of("red"));]
  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 \texttt{CPoint.sum} resets the color to \texttt{red}: we call the \texttt{of(int, int)} method, that now delegates to \texttt{of(int, int, Color)} by passing \texttt{red} as the default field value. What should be the behaviour in this case? If our abstract state supports withers, we can use \texttt{this.withX(newX).withY(newY)}, instead of writing \texttt{This.of(...)}, in order to preserve the color from \texttt{this}. This solution is better but still not satisfactory since the color from \texttt{that} is ignored.

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

```java
p = {
  method int x() method int y() //getters
  method This withX(int that) method This withY(int that) //withers
  static method This of(int x, int y)
  method This merge(This that) //new method merge!
}
pointSum = Use p, {
  method This sum(This that) {
    return this.merge(that).withX(this.x()+that.x()).withY(this.y()+that.y());
  }
  colored = {
    method Color color()
    method This withColor(Color that)
    method This merge(This that) {
      //how to merge color handled here
      return this.withColor(this.color().mix(that.color()));
    }
  }
}
CPoint = /*as before*/
```

Independent Extensibility: Of course, quite frequently there can be multiple independent extensions \cite{41} that need to be composed. Lets suppose that we could have a notion of \texttt{flavored} as well. In order to compose \texttt{colored} with \texttt{flavored} we would need to compose their respective merge operations. To this aim \texttt{Use} is not sufficient. To combine the implementation of two different implementation of methods, we introduce an operator called \texttt{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 \texttt{flavored} trait:

```java
flavored = {
  method Flavor flavor() //very similar to colored
  method This withFlavor(Flavor that)
  method This merge(This that) {
    //merging flavors handled here
    return this.withFlavor(this.flavor().mix(that.flavor()));
  }
}
```

In order to merge \texttt{colored} and \texttt{flavored} we use \texttt{super} to introduce method selectors \texttt{_1merge} and \texttt{_2merge} to refer to the version of \texttt{merge} as defined in the first/second element of \texttt{Use}.
Separating Use and Reuse to Improve Both

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

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.

```java
p = /* as originally designed: no merge */
pointSum = /* as originally designed: no merge */
merge = {method This merge(This that)}
pointSumMerge = Use merge, pointSum[super sum as _1sum], {
  method This sum(This that){return this.merge(that)._1sum(that);}}
colored = /* as before, with merge implementation */
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\(^6\) 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.

\(^6\) 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 CPPoint and CFPoint classes (12 + 18). The major benefit of 42\(\mu\) is the reduction of the amount of glue-code needed to generate CPPoint and CFPoint (4 + 9).

The results for the second experiment are presented below.

<table>
<thead>
<tr>
<th>Language</th>
<th>Lines of code</th>
<th>N. of members</th>
<th>N. of classes or traits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>47 = 10 + 9</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td>Scala</td>
<td>53 = 8 + 3 + 6 * 2 + 12 + 18</td>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>42(\mu)</td>
<td>33 = 7 + 3 + 5 * 2 + 4 + 9</td>
<td>24</td>
<td>6</td>
</tr>
</tbody>
</table>

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:

```java
abstract class A{
    static class B{..}
    abstract B m();
}
class AA extends A{
    static class B{..} B m(){..}}//Error: Invalid overriding
```

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.

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:

```java
t={
    method B m() B={..}
  }
D= t[rename B into C]
```

would flatten to:

```java
t={
    method B m() B={..}
  }
D= t[rename B into C]=
  D={
    method C m() C={..}
  }
```

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 L42 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:

```
valPlus= Use plus, {//eval operation for Plus data-variant
    Exp= {interface
        method int eval()
    }
    Plus= {implements Exp
        method int eval(){
            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:

```
exp= { Exp= {interface
        method int eval()}
    }
num= Use exp[rename T into Num],{/T is renamed to Num and summed with
    Num= {method int value() static method Num of(int value)}} // this Num
plus= Use exp[rename T into Plus], {
    Plus= {method Exp left() method Exp right()
        static method Plus of(Exp left,Exp right)}
}
```

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.

```
eval= {Exp= {interface
        method int eval()}
    }
evalNum= Use num, eval[rename T into Num],{/just the implementation
    Num= { method int eval(){return this.value();} }});//of the specific method
```
evalPlus = Use plus, eval[rename T into Plus], {
  Plus={ method int eval(){ return this.left().eval()+this.right().eval();} }}

evalNeg = Use neg, eval[rename T into Neg], {Neg={ method int eval(){..}}}
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 (\(\text{eval}, \text{show}, \text{double}\)) we just needed 4 lines to declare the operation in the interface, and 2 lines for each of the 3 data-variants.

Following [41], after \text{double} we present an implementation of \text{equals}. Their solution involved double dispatch to avoid casting. To show understandable code, we show a simpler solution with a guarded cast (sometimes called a typecase).\(^7\) The idea is that since every data-variant contains the same "cast" logic, we can modularize it into an \text{equals} trait; \text{equals} in [41] is complex and requires glue code.

```scala
equals= {
  Exp= {interface method Bool equals(Exp that)}
  T= {implements Exp
      method Bool exactEquals(T that)
      method equals(that){
        if(T instanceof This){return this.exactEquals(that);}else{ return false;}}}
}
```

The Scala code here can be made fully isolated with little extra syntactic cost. The original Scala \text{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.

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 \text{Sum}={\text{method} Bool equalToNum(Num that){\text{return false;}}}).

\(^7\) The interested reader can find a \(42_\mu\) implementation of \text{equals} with double dispatch in the appendix.
6 Summary of formalisation

In Appendix A we formalise $\mu$; 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 ensures 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 $\mu$ 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:

```java
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).

$\mu$ 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 subtyping 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 $\mu$ 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 three 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. $\mu$ relies on DJ’s approach but, thanks to our novel representation of state, $\mu$ 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).

- **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:

```java
class A {
    static class B {
        int f1;
    }
    int k(.B x) {
        return x.f1;
    }
}

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:

```java
new AA().k(new AA.B()) // well-typed
new A().k(new A.B()) // well-typed
A a = new AA(); // well-typed assuming AA is a subtype of A
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. 42 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 42, this also reduces the expressive power a little, but in the full 42 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.
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.

<table>
<thead>
<tr>
<th></th>
<th>direct instantiation</th>
<th>self instantiation</th>
<th>unit of use</th>
<th>unit of reuse</th>
<th>introduces a type</th>
<th>induced type is the type of this</th>
<th>inheritance induces subtyping</th>
<th>typed before imported</th>
<th>typed before composed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java/Scala classes</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Java8 interfaces</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Scala traits</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Original traits</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>TR</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>42$_\mu$ traits</td>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>42$_\mu$ classes</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>Module composition</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>DJ classes</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>X</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>PT</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

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$_\mu$ 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


[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.


[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.


Separating Use and Reuse to Improve Both


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: \texttt{This.Exp}, \texttt{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.\footnote{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.}

We also consider a binary operator sum ($+$) instead of the nary operator \texttt{Use}. Figure 1 contains the complete formalization for $42_\mu$: syntax, compilation process, typing, and finally reduction.

\section*{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: $\cdot$. This operation is a generalization to the case of nested classes of the simplest and most elegant trait composition operator \cite{eight.oldstyle}. 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 an interface class, while a lowercase one is an 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 \texttt{implements} keyword.

Method declarations $MD$ can be instance methods or \texttt{static} methods. A static method in $42_\mu$ is similar to a \texttt{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 $\nu_D$ are just calls to abstract static methods: thanks to abstract state, we have no \texttt{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.
Separating Use and Reuse to Improve Both

\[
\begin{align*}
ID & ::= t \mid C & \text{class or trait name} \\
DE & ::= ID = E & \text{Meta-declaration} \\
D & ::= ID = L & \text{Declaration} \\
E & ::= L \mid t \mid E \cdot E \mid \ldots & \text{Code Expression} \\
L & ::= \{\text{interface? implements } T \ M \} & \text{Code Literal} \\
T & ::= C \mid C.T & \text{Type} \\
M & ::= \text{static? method } T \ m(Tx) \ e? \mid C = L & \text{Member} \\
e & ::= x \mid e.m(\overline{e}) \mid T.m(\overline{e}) & \text{Expression} \\
v_\pi & ::= T.m(\nu_\pi), \text{where } m \text{ is abstract in } D(T) & \text{value} \\
\mathcal{E}_\pi & ::= \nu_\pi, m(\overline{\nu_\pi}, e_\pi) \mid T.m(\nu_\pi, e_\pi, \overline{e}) & \text{evaluation context} \\
\mathcal{E}_c & ::= \nu_\pi, m(\overline{\nu_\pi}, e_\pi) \mid T.m(\nu_\pi, e_\pi, \overline{e}) & \text{compilation context} \\
\mathcal{E} & ::= \nu_\pi, m(\overline{\nu_\pi}, e_\pi) \mid T.m(\nu_\pi, e_\pi, \overline{e}) & \text{ctx} \\
\Gamma & ::= x_1:T_1, \ldots, x_n:T_n & \text{variable environment}
\end{align*}
\]

\[
\begin{align*}
(\text{TOP}) & \quad E_0 \xrightarrow{\pi} E_1 & \text{VAR}\in D, D \vdash D : OK \\
(\text{LOOK-UP}) & \quad D \xrightarrow{ID=E} \xrightarrow{DE} D \xrightarrow{D \vdash D(t)} \xrightarrow{E_0 \vdash E_1} \xrightarrow{\mathcal{E}_0 \vdash \mathcal{E}_1} \xrightarrow{L \xrightarrow{L_1+L_2}} L = L_1 + L_2 \\
(\text{SUM}) & \quad E_0 \xrightarrow{\pi} E_1 & \text{VAR}\in D, D \vdash D : OK \\
(\text{TD-OK}) & \quad T \vdash t \cdot L : OK & \text{VAR}\in D, D \vdash T \leq T \\
(\text{L-OK}) & \quad \forall M \in \mathcal{M}, T ; D \vdash M : OK & \text{VAR}\in D, D \vdash \{\_implies \text{implements } T \ M \} : OK \\
(\text{CD-OK}) & \quad C, D \vdash L_1 \vdash L_0 : OK & \text{VAR}\in D, D \vdash \text{coherent}(C, L_1) \\
(\text{NESTED-OK}) & \quad T ; D \vdash C ; L : OK & \text{VAR}\in D, D \vdash \{\_implies \text{implements } T \ M \} : OK \\
(\text{SUBTYPE}) & \quad T \vdash T_1 \leq T_2 & \text{VAR}\in D, D \vdash \{\_implies \text{implements } T \ M \} : OK \\
(\text{S-REFL}) & \quad T \vdash t \cdot L : OK & \text{VAR}\in D, D \vdash \{\_implies \text{implements } T \ M \} : OK \\
(\text{METHOD-OK}) & \quad \text{if } e? = e \text{ then } D \vdash \Gamma \vdash e : T_0 & \text{VAR}\in D, D \vdash \text{if } \text{static? } = \text{ static } \text{ then } \Gamma = x_1 : T_1 \ldots x_n : T_n \\
& \quad \text{else } \Gamma = \text{this} : T, x_1 : T_1 \ldots x_n : T_n \\
(\text{SUBSUMPTION}) & \quad D ; \Gamma \vdash e : T_1 \\
& \quad D \vdash T_1 \leq T_2 \\
& \quad D ; \Gamma \vdash e : T_2 & \text{VAR}\in D, D \vdash \text{static-method-call} \\
(\text{STATIC-METHOD-CALL}) & \quad D ; \Gamma \vdash e : T_1 \ldots T_2 ; \Gamma \vdash e_n : T_n & \text{VAR}\in D, D \vdash \text{static-method } T \ m(T_1x_1 \ldots T_2x_n) \ e? : OK \\
(\text{x}) & \quad D ; \Gamma \vdash x : \Gamma(x) & \text{VAR}\in D, D \vdash \text{method } T \ m(T_1x_1 \ldots T_2x_n) \ e? : OK \\
(\text{METHOD-CALL}) & \quad D ; \Gamma \vdash e_0 : T_0 \ldots T_1 ; \Gamma \vdash e_n : T_n & \text{VAR}\in D, D \vdash \text{method } T \ m(T_1x_1 \ldots T_2x_n) \ e? : OK \\
(\text{CTXV}) & \quad e_0 \xrightarrow{\pi} e_1 & \text{VAR}\in D, D \vdash \text{s-M} \\
(\text{S-M}) & \quad T ; m(\nu_\pi) \rightarrow \text{meth}(D(T, m), \nu_\pi) & \text{VAR}\in D, D \vdash \text{meth}(D(T, m), \nu_\pi, \nu_\pi) \nu_\pi = T.m(\_)
\end{align*}
\]

\bf{Figure 1} Formalization

28
A.2 Well-formedness

The whole program ($\mathcal{DE}$) is well formed if all the traits and classes at top level have unique names. The special class name \texttt{This} is not one of those, and the subtype relations are consistent: this means that the implementation of interfaces is not circular, and that $\forall \, \text{ID} \in \mathcal{E}[L] \in \mathcal{DE}, \text{consistentSubtype}(\mathcal{DE}, \text{This}=L; L)$. 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 $\text{consistentSubtype}(\mathcal{DE}; L)$

- $\text{consistentSubtype}(\mathcal{DE}, \{\text{interface}\,?\,\text{implements}\,\mathcal{T}\})$ where
  \[\forall\, \mathcal{T} \in \mathcal{T}, \, \mathcal{DE}(\mathcal{T}) = \{\text{interface}\}_{\mathcal{T}},\]
  \[\forall\, \_ = L \in \mathcal{M}, \, \text{consistentSubtype}(\mathcal{DE}; L) \text{ and}\]
  \[\forall m, T \in \mathcal{T}, \, \text{if method } T_0 m(Tx) \in \mathcal{DE}(T) \text{ then method } T_0 m(Tx) e? \in \mathcal{M}\]

A code literal $L$ is well formed iff:

- for all methods: parameters have unique names and no parameter is named \texttt{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 \texttt{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 $\text{top}$, $\text{look-up}$, $\text{ctx-c}$ and $\text{sum}$. To model more composition operators, they would each need their own rule.

Rule $\text{top}$ compiles the leftmost top level (trait or class) declaration that needs to be compiled. First it identifies the subset of the program $\mathcal{D}$ 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 $\text{look-up}$): $\forall t, \text{if } E = \varepsilon[t] \text{ then } t \in \text{dom}(\mathcal{D})$. If the required $\mathcal{D}$ cannot be typed, this would cause a compilation error at this stage. Rule $\text{look-up}$ replaces a trait name $t$ with the corresponding literal $L$. Since $\mathcal{D}$ is all well typed, $L$ is well typed too. Rule $\text{ctx-c}$ uses the compilation context to apply a deterministic left to right call by value reduction; thus the leftmost invalid sum that is performed will be the one providing the compilation error.

---

9 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.

10 In the flattening process, values are code literals $L$. © 2023 Arora, Servetto, Oliveira
Separating Use and Reuse to Improve Both

Keeping in mind the order of members in a literal is immaterial, rule \texttt{sum} applies the operator:

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

- $L_1 + L_2 = L_3$ where
  
  $L_1 = \{\text{interface? implements } T_1 M_1 \overline{M}_0\}$
  
  $L_2 = \{\text{interface? implements } T_2 \overline{M}_2 \overline{M}_0\}$
  
  $L_3 = \{\text{interface? implements } T_1, T_2 \overline{M}_1, \overline{M}_2 (\overline{M}_0 + \overline{M}_0)\}$

  $\text{dom}(\overline{M}_1)$ disjoint $\text{dom}(\overline{M}_2)$ and $\text{dom}(\overline{M}_0) = \text{dom}(\overline{M}_0)$

- $(M_1 \ldots M_n) + (M'_1 \ldots M'_n) = (M_1 + M'_1) \ldots (M_n + M'_n)$

- $M_1 + M_2 = M_2 + M_1$

- $C * L_1 + C * L_2 = C * L_3$ if $L_1 + L_2 = L_3$

- \textbf{static? method} $T_0 m(\overline{T}x)$ + \textbf{static? method} $T_0 m(\overline{T}x)e$? = \textbf{static? method} $T_0 m(\overline{T}x)e$?

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

- \textbf{Method-clash}: two methods with the same name are composed, but either their headers have different types or they are both implemented.

- \textbf{Class-clash}: a class is composed with an interface.\(^1\)

- \textbf{Implements-clash}: the resulting code would not be well formed. For example, in the following $t_1 + t_2$ would result in a class $B$ implementing $A$ with method $a()$, but $B$ does not have such method.\(^2\)

\begin{verbatim}
1 t1={ A= {interface method Void a()}}
2 t2={ A= {interface} B= {implements A}}
\end{verbatim}

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 $\text{consistentSubtype}(D,E,L)$.

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 \texttt{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 \texttt{This} to the typed program.

\(^1\) The full language relaxes this condition, for example an empty class can be seen as an empty interface during composition.

\(^2\) In 42\(_m\) 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$(T,\{\text{interface? implements} T \bar{M}\})$ holds where
  
  $\forall C=L' \in \bar{M} \text{coherent}(T,C,L')$

  and either interface? = interface

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

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 method $T' m(\overline{TX}) \in \bar{M}$ we select only abstract methods.

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

- factory$(T,M_1\ldots M_n) = M_i = \text{static method } T m(_\_)$ where
  
  $\forall j \neq i$. $M_j$ is not of the form static method $\_\_(_\_)$

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

Define state$(M,M')$

- state(static method $T m(T_1x_1\ldots T_nx_n)$, method $T_i x_i()$)

- state(static method $T m(T_1x_1\ldots T_nx_n)$, 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 $\vdash D : OK$, $e$ not of form $v_\$ and $\overline{D} \vdash e : T$ then $e \leftarrow_{\overline{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 [three, seven]; here we can obtain the same result in a much simpler setting. We denote wrong$(\overline{D},E)$ to be the number of $L$s such that $E = \emptyset[L]$ and not $\overline{D} \vdash L : OK$.

**Theorem A.2** (Compilation Soundness). if $E_0 \rightarrow_{\overline{\pi}} E_1$ then wrong$(\overline{D},E_0) \geq \text{wrong}(\overline{D},E_1)$.

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

- look-up preserves the number of wrong literals: $t \in \overline{D}$ and $\overline{D}$ 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 (1-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 \textsc{sum} is applied the result still respect \textsc{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 $L_1 \ldots L_n$, and after successful flattening $C = L$ can not be type-checked, then the issue was originally present in one of $L_1 \ldots L_n$. This also means that as an optimization strategy we may remember what method bodies come from traits and what method bodies come from code literals, 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:

```java
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:

```java
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 \textsc{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:

```java
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\textsuperscript{13} between multiple class declarations without the need of predicting the resulting shape.\textsuperscript{14}

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:

```plaintext
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 $\top$, $D = ta$ and $D' = tc$). 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 $\text{SUM}$ rule could not be performed (for example if $tc$ had a method $\text{hello}$ too) a composition error would be generated at this stage. On the other hand, if $B$ and $C$ were swapped, as in:

```plaintext
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:

```plaintext
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-

\textsuperscript{13} OO languages leverage on recursive types most of the times: for example $\text{String}$ may offer a $\text{Int}$ $\text{size()}$ method, and $\text{Int}$ may offer a $\text{String}$ $\text{toString()}$ method. This means that typing classes $\text{String}$ and $\text{Int}$ in isolation one at a time is not possible.

\textsuperscript{14} This is needed in full $42\mu$: it is impossible to predict the resulting shape since arbitrary code can run at compile time.
Separating Use and Reuse to Improve Both

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 $\text{meth}$:

Define $\text{meth}(M, v_D)$

- $\text{meth}(\text{static method } T m(T_1 x_1 \ldots T_n x_n) e, v_{\sigma_1} \ldots v_{\sigma_n}) = e[x_1 = v_{\sigma_1}, \ldots, e_n = v_{\sigma_n}]$
- $\text{meth}(\text{method } T m(T_1 x_1 \ldots T_n x_n) e, v_{\sigma_0}, v_{\sigma_1}, \ldots, v_{\sigma_n}) = e[\text{this } = v_{\sigma_0}, x_1 = v_{\sigma_1}, \ldots, e_n = v_{\sigma_n}]$
- $\text{meth}(\text{method } T_i x_i(), T.m(v_{\sigma_1} \ldots v_{\sigma_n})) = v_{\sigma_i}$
  
  where $D(T, m) = \text{static method } T m(T_1 x_1 \ldots T_n x_n)$
- $\text{meth}(\text{method } T \text{ with } x_i(T_i \text{ that }), T.m(v_{\sigma_1} \ldots v_{\sigma_n}) v_{\sigma}) = T.m(v_{\sigma_1} \ldots v_{\sigma_{i-1}}, v_{\sigma}, v_{\sigma_{i+1}} \ldots v_{\sigma_n})$
  
  where $D(T, m) = \text{static method } T m(T_1 x_1 \ldots T_n x_n)$

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:

```java
// one.osf
reusable Set={...}
// expands into
// two.osf
set={...}
// three.osf
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.¹⁵

- **(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

![Image](image-url)

---

¹⁵ 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:

```
//one.tosf
class A{
  int ma() { return Utils.m(this); }
}

//two.tosf
class Utils{
  static int m(A a) {...}
}

//three.tosf
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:

```
//one.tosf
interface IA{
  interface method int ma();}

//two.tosf
class Utils{
  static method int m(IA a) {...}
}

//three.tosf
class A implements IA{
  method int ma() { return Utils.m(this); }}

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

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

```
//one.tosf
IA={interface method int ma()}; //type = interface

//two.tosf
Utils={static method int m(IA a){return ... }}

ta={implements IA
  method int ma(){return Utils.m(this);}}

//three.tosf
A=Use ta,

//four.tosf
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:

```
//class A{
public int x;
public A(String s){
  this.x=/*any computation here*/;
}
}
```

could be expressed in $42\mu$ as
A={
  method int x();
  static method This of(int x)
  static method This of(String s){
    return This.of(/*any computation here*/);
  }
}

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:

t={
  method int x();
  method void x(int x);
  method int defaultX(){return 0;} // new concept here!
}

would loosely correspond to a Java abstract class:

abstract class T{
  private int x=0;
  public int x(){return x;}
  public void x(int that){return this.x=that;}
}

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:

A= Use t,{ static method This of()}
B= Use t,{ method String s(); static method This of(String s)}
...
A a=A.of();
a.x()==0; // assert a.x()==0
a.x(42);
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, 42μ 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 42μ, 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\textsubscript{µ} one. However, when it comes to model complex reuse scenarios, we believe 42\textsubscript{µ} 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\textsubscript{µ} requires being explicit when subtyping and subclassing are both desired by requiring an extra interface.
- \texttt{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

```java
//case 0 of 16
class Point {//6 lines
  final int x;
  final int y;
  public Point(int x, int y) {
```

38
Hrshikesh Arora, Marco Servetto, and Bruno C. d. S. Oliveira

```java
this.x = x;
this.y = y;
}
}
// 1
class PointSum extends Point {//5*4
public PointSum(int x, int y) {
super(x, y);
}
public PointSum sum(Point that) {
return new PointSum(this.x + that.x, this.y + that.y);
}
// 2
class PointSub extends Point {
public PointSub(int x, int y) {
super(x, y);
}
public PointSub sub(Point that) {
return new PointSub(this.x - that.x, this.y - that.y);
}
// 3
class PointMul extends Point {
public PointMul(int x, int y) {
super(x, y);
}
public PointMul mul(Point that) {
return new PointMul(this.x * that.x, this.y * that.y);
}
// 4
class PointDiv extends Point {
public PointDiv(int x, int y) {
super(x, y);
}
public PointDiv div(Point that) {
return new PointDiv(this.x / that.x, this.y / that.y);
}
// 5
class PointSumSub extends PointSum {//7*6
public PointSumSub(int x, int y) {
super(x, y);
}
public PointSumSub sum(Point that) {
return new PointSumSub(this.x + that.x, this.y + that.y);
}
public PointSumSub sub(Point that) {
return new PointSumSub(this.x - that.x, this.y - that.y);
}
// 6
class PointSumMul extends PointSum {
public PointSumMul(int x, int y) {
super(x, y);
}
public PointSumMul sum(Point that) {
return new PointSumMul(this.x + that.x, this.y + that.y);
}
public PointSumMul mul(Point that) {
return new PointSumMul(this.x * that.x, this.y * that.y);
}
// 7
class PointSumDiv extends PointSum {
public PointSumDiv(int x, int y) {
super(x, y);
}
public PointSumDiv sum(Point that) {
return new PointSumDiv(this.x + that.x, this.y + that.y);
}
public PointSumDiv div(Point that) {
return new PointSumDiv(this.x / that.x, this.y / that.y);
}
```
Separating Use and Reuse to Improve Both

```java
// 8
class PointSubMul extends PointSub {
    public PointSubMul(int x, int y) {
        super(x, y);
    }
    public PointSubMul sub(Point that) {
        return new PointSubMul(this.x - that.x, this.y - that.y);
    }
    public PointSubMul mul(Point that) {
        return new PointSubMul(this.x * that.x, this.y * that.y);
    }
}

// 9
class PointSubDiv extends PointSub {
    public PointSubDiv(int x, int y) {
        super(x, y);
    }
    public PointSubDiv sub(Point that) {
        return new PointSubDiv(this.x - that.x, this.y - that.y);
    }
    public PointSubDiv div(Point that) {
        return new PointSubDiv(this.x / that.x, this.y / that.y);
    }
}

// 10
class PointMulDiv extends PointMul {
    public PointMulDiv(int x, int y) {
        super(x, y);
    }
    public PointMulDiv mul(Point that) {
        return new PointMulDiv(this.x * that.x, this.y * that.y);
    }
    public PointMulDiv div(Point that) {
        return new PointMulDiv(this.x / that.x, this.y / that.y);
    }
}

// 11
class PointSumSubDiv extends PointSumSub {
    public PointSumSubDiv(int x, int y) {
        super(x, y);
    }
    public PointSumSubDiv sum(Point that) {
        return new PointSumSubDiv(this.x + that.x, this.y + that.y);
    }
    public PointSumSubDiv sub(Point that) {
        return new PointSumSubDiv(this.x - that.x, this.y - that.y);
    }
    public PointSumSubDiv div(Point that) {
        return new PointSumSubDiv(this.x / that.x, this.y / that.y);
    }
}

// 12
class PointSumMulDiv extends PointSumMul {
    public PointSumMulDiv(int x, int y) {
        super(x, y);
    }
    public PointSumMulDiv sum(Point that) {
        return new PointSumMulDiv(this.x + that.x, this.y + that.y);
    }
    public PointSumMulDiv sub(Point that) {
        return new PointSumMulDiv(this.x - that.x, this.y - that.y);
    }
    public PointSumMulDiv mul(Point that) {
        return new PointSumMulDiv(this.x * that.x, this.y * that.y);
    }
}
```

40
public PointSumMulDiv sum(Point that) {
    return new PointSumMulDiv(this.x + that.x, this.y + that.y);
}

public PointSumMulDiv mul(Point that) {
    return new PointSumMulDiv(this.x * that.x, this.y * that.y);
}

public PointSumMulDiv div(Point that) {
    return new PointSumMulDiv(this.x / that.x, this.y / that.y);
}

C.1.2 Classless Java

interface Point {
    static Point of(int x, int y) { // "of" methods are generated by @Obj) thus their lines are not counted
        return new Point() {
            public int x() {
                return x;
            }
            public int y() {
                return y;
            }
        };
    }
}
// 9
interface PointSum extends Point {
    static PointSum of(int x, int y) { // "of" methods are generated by @Obj) thus their lines are not counted
        return new PointSum() {
            public int x() {
                return x;
            }
            public int y() {
                return y;
            }
        };
    }
}
Separating Use and Reuse to Improve Both

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

Separating Use and Reuse to Improve Both

```java
public int x() {
    return this.x();
}

public int y() {
    return this.y();
}

default PointSubDiv div(Point that) {
    return of(this.x() / that.x(), this.y() / that.y());
}

default PointMulDiv mul(Point that) {// we have to rewrite the method to call the new "of"
    return of(this.x() * that.x(), this.y() * that.y());
}

default PointSumMulDiv div(Point that) {
    return of(this.x() / that.x(), this.y() / that.y());
}

interface PointMulDiv extends PointMul, PointDiv {
    static PointMulDiv of(int x, int y) {
        return new PointMulDiv() {
            public int x() {
                return x;
            }
            public int y() {
                return y;
            }
        };
    }

    default PointMulDiv mul(Point that) {// we have to rewrite the method to call the new "of"
    return of(this.x() * that.x(), this.y() * that.y());
}

    default PointMulDiv div(Point that) {
        return of(this.x() / that.x(), this.y() / that.y());
    }
}

interface PointSumSubDiv extends PointSumSub, PointSumDiv, PointSubDiv {
    static PointSumSubDiv of(int x, int y) {
        return new PointSumSubDiv() {
            public int x() {
                return x;
            }
            public int y() {
                return y;
            }
        };
    }

    default PointSumSubDiv sum(Point that) {
        return of(this.x() + that.x(), this.y() + that.y());
    }

    default PointSumSubDiv sub(Point that) {
        return of(this.x() - that.x(), this.y() - that.y());
    }

    default PointSumSubDiv div(Point that) {
        return of(this.x() / that.x(), this.y() / that.y());
    }
}

interface PointSumSubMul extends PointSumSub, PointSumMul, PointSubMul {
    static PointSumSubMul of(int x, int y) {
        return new PointSumSubMul() {
            public int x() {
                return x;
            }
            public int y() {
                return y;
            }
        };
    }

    default PointSumSubMul sum(Point that) {
        return of(this.x() + that.x(), this.y() + that.y());
    }

    default PointSumSubMul sub(Point that) {
        return of(this.x() - that.x(), this.y() - that.y());
    }

    default PointSumSubMul mul(Point that) {
        return of(this.x() * that.x(), this.y() * that.y());
    }
}

interface PointSumMulDiv extends PointSumMul, PointMulDiv, PointSumDiv {
```
Hrshikesh Arora, Marco Servetto, and Bruno C. d. S. Oliveira

C.1.3 Scala

trait tPointState {
  // 5
  interface PointSubMulDiv extends PointSubMul, PointMulDiv, PointSubDiv {
    static PointSubMulDiv of(int x, int y) {
      return new PointSubMulDiv() {
        public int x() {
          return x;
        }
        public int y() {
          return y;
        }
      };
    }
    default PointSubMulDiv sum(Point that) {
      return of(this.x() + that.x(), this.y() + that.y());
    }
    default PointSubMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
      return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
    }
    default PointSubMulDiv div(Point that) {
      return of(this.x() / that.x(), this.y() / that.y());
    }
  }
}

trait tPointState {
  // 14
  interface PointSubMulDiv extends PointSubMul, PointMulDiv, PointSubDiv {
    static PointSubMulDiv of(int x, int y) {
      return new PointSubMulDiv() {
        public int x() {
          return x;
        }
        public int y() {
          return y;
        }
      };
    }
    default PointSubMulDiv sub(Point that) {
      return of(this.x() - that.x(), this.y() - that.y());
    }
    default PointSubMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
      return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
    }
    default PointSubMulDiv div(Point that) {
      return of(this.x() / that.x(), this.y() / that.y());
    }
  }
}

trait tPointState {
  // 15
  interface PointSubMulDiv extends PointSubMul, PointSumMulDiv, PointSubDiv, PointMulDiv {
    static PointSubMulDiv of(int x, int y) {
      return new PointSubMulDiv() {
        public int x() {
          return x;
        }
        public int y() {
          return y;
        }
      };
    }
    default PointSubMulDiv sum(Point that) {
      return of(this.x() + that.x(), this.y() + that.y());
    }
    default PointSubMulDiv sub(Point that) {
      return of(this.x() - that.x(), this.y() - that.y());
    }
    default PointSubMulDiv mul(Point that) { // we have to rewrite the method to call the new "of"
      return of(this.x() * that.x(), this.y() * that.y()); // in order to produce an instance of PointSumMul
    }
    default PointSubMulDiv div(Point that) {
      return of(this.x() / that.x(), this.y() / that.y());
    }
  }
}
Separating Use and Reuse to Improve Both

```scala
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
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 }
```
C.1.4 42

The main differences with respect to 42 as presented in the paper are: we use `Resource<T>` 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.

```scala
class PointSumSubDiv(val x : Int, val y : Int) extends tPointSum with tPointSub with tPointDiv {
  override type p = PointSumSubDiv
  override def of(x : Int, y : Int) = new PointSumSubDiv(x,y)
}

class PointSumSubMul(val x : Int, val y : Int) extends tPointSum with tPointSub with tPointMul {
  override type p = PointSumSubMul
  override def of(x : Int, y : Int) = new PointSumSubMul(x,y)
}

class PointSumMulDiv(val x : Int, val y : Int) extends tPointSum with tPointMul with tPointDiv {
  override type p = PointSumMulDiv
  override def of(x : Int, y : Int) = new PointSumMulDiv(x,y)
}

class PointSubMulDiv(val x : Int, val y : Int) extends tPointSub with tPointMul with tPointDiv {
  override type p = PointSubMulDiv
  override def of(x : Int, y : Int) = new PointSubMulDiv(x,y)
}

class PointSumSubMulDiv(val x : Int, val y : Int) extends tPointSum with tPointSub with tPointMul with tPointDiv {
  override type p = PointSumSubMulDiv
  override def of(x : Int, y : Int) = new PointSumSubMulDiv(x,y)
}
```

```scala
$pointSum: Resource<>< () { //1*16
  method Num x() 
  method Num y()
  class method This of(Num x, Num y)
}

$pointSub: Resource<>< () {
  method This sub(Num x, Num y)
  This of(x: this.x()+that.x(), y: this.y()+that.y())
}

$pointMul: Resource<>< () {
  method This mul(Num x, Num y)
  This of(x: this.x()*that.x(), y: this.y()*that.y())
}

$pointDiv: Resource<>< () {
  method This div(Num x, Num y)
  This of(x: this.x()/that.x(), y: this.y()/that.y())
}

Point: Use[pointSum]() <> {}

PointSum: Use[pointSum]() <> {}
PointSub: Use[pointSub]() <> {}
PointMul: Use[pointMul]() <> {}
PointDiv: Use[pointDiv]() <> {}

PointSumSub: Use[pointSum]; pointSub]() <> {}
PointSumMul: Use[pointSum]; pointMul]() <> {}
PointSumDiv: Use[pointSum]; pointDiv]() <> {}
PointSubMul: Use[pointSub]; pointMul]() <> {}
PointSubDiv: Use[pointSub]; pointDiv]() <> {}

47
```
Separating Use and Reuse to Improve Both

C.2 FPoint

C.2.1 Java

```java
enum Flavor {
    NONE, SOUR, SWEET, SALTY, SPICY;
}

class Color {
    final int r;
    final int g;
    final int b;
    public Color(int r, int g, int b) {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    public Color mix(Color that) {
        return new Color((this.r + that.r) / 2, (this.g + that.g) / 2, (this.b + that.b) / 2);
    }
}

class Point {//10 lines
    final int x;
    final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point withX(int that) {
        return new Point(that, this.y);
    }
    public Point withY(int that) {
        return new Point(this.x, that);
    }
    public Point sum(Point that) {
        return this.withX(this.x + that.x).withY(this.y + that.y);
    }
}

class CPoint extends PointSum {//13 lines
    final Color color;
    public CPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }
    public CPoint withX(int that) {
        return new CPoint(that, this.y, this.color);
    }
    public CPoint withY(int that) {
```
```java
return new CPoint(this.x, that,
    this.color);
}
}
//class FCPoint extends CPoint {//15 lines
final Flavor flavor;
public FCPoint(int x, int y, Color color, Flavor flavor) {
    super(x, y, color);
    this.flavor = flavor;
}
public FCPoint with(int that) {
    return new FCPoint(that,
        this.y,
        this.color,
        this.flavor);
}
public FCPoint withY(int that) {
    return new FCPoint(this.x, that,
        this.color,
        this.flavor);
}
public FCPoint withColor(Color that) {
    return new FCPoint(this.x,
        this.y, that,
        this.color);
}
public FCPoint withFlavor(Flavor that) {
    return new FCPoint(this.x,
        this.y,
        this.color, that);
}
public FCPoint merge(FCPoint that) {
    return this.withColor(that.color).withFlavor(that.flavor);
}
}
C./two.osf./two.osf

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 trait tPointSum extends tPointState {//3
11 def sum(that:p)=
12   this.merge(that).withX(this.x+that.x).withY(this.y+that.y)
13 }
14 }
15 trait tColored {//6*2
16 type p <: tColored
17 def color:Int
18 def withColor(that:Int):p
19 def merge(that:p)=
20   this.withColor(this.color+that.color)
21 }
22 }
23 trait tFlavored{
24 type p <: tFlavored
25 def flavor:Int
26 def withFlavor(that:Int):p
27 def merge(that:p)=
28   this.withFlavor(that.flavor)
29 }
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

C.2.3 42

1 Flavor: Enumeration"sour, sweet, salty, spicy"
2
3 Color: Data<> { Num r Num g Num b
4
5 method This mix(This that)
6 Color r :(this.r()+that.r())/2Num, g:(this.g()+that.g())/2Num, b:(this.b()+that.b())/2Num
7 }
8
9 $p: Resource <> //{ 7 lines
10 method Num x()
11 method Num y() // getters
12 method This withX(Num that)
13 method This withY(Num that) // withers
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 <> {// 3 lines
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(); $colored() <> {// 4 lines
37 class method This of(Num x, Num y)
38 This.of(x:x, y:y, color:Color(r:100Num, g:0Num, b:0Num))
39 class method This of(Num x, Num y, Color color)
40 }
41
42 FCPoint: Use //9 lines
43 Refactor2.toAbstract{selector:"merge(that)", into:"_1merge(that)", into:"_2merge(that)"}<|$colored|};
44 Refactor2.toAbstract{selector:"merge(that)", into:"_1merge(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.

trait Base {

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

```scala
92 class Neg(t: exp) extends super.Neg(t) with NegBehavior with Exp
93 94 --/-- 52 lines up to here, not counting new lines and '}'
95 trait Equals extends Base {
96 | type exp <% Exp;
97 |
98 | trait Exp extends super.Exp {
99 | \[def eql(other: exp): Boolean;
100 | 101 | def isNum(v: Int): Boolean = false;
102 | 103 | trait NumBehavior extends Exp {
104 | \[self: BaseNum =>
105 | 106 | def eql(other: exp): Boolean = other.isNum(value);
107 | 108 | override def isNum(v: Int) = v == value;
109 | 110 } final class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
111 } final class Num(v: Int) extends BaseNum(v) with NumBehavior // effectively super[Equals].NumBehavior
112 113 114 trait EqualsPlusNeg extends BasePlusNeg with Equals {
115 | type exp <% Exp;
116 |
117 | trait Exp extends super[BasePlusNeg].Exp {
118 | \[def isPlus(l: exp, r: exp): Boolean = false;
119 | 120 | trait PlusBehavior extends Exp {
121 | \[self: BasePlus =>
122 | 123 | def eql(other: exp): Boolean = other.isPlus(left, right);
124 | 125 | override def isPlus(l: exp, r: exp) = (left eql l) && (right eql r)
126 } final class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
127 |
128 | trait NegBehavior extends Exp {
129 | \[def eql(other: exp): Boolean = other.isNeg(term);
130 | 131 | override def isNeg(t: exp) = term eql t
132 } final class Neg(term: exp) extends BaseNeg(term) with NegBehavior with Exp
133 |
134 | trait EqualsShowPlusNeg extends EqualsPlusNeg with EqualsPlusNeg {
135 | type exp <% Exp;
136 |
137 | trait Exp extends super[EqualsPlusNeg].Exp {
138 | \[def isPlus(l: exp, r: exp): Boolean = (left eql l) && (right eql r)
139 | 140 | trait PlusBehavior extends Exp {
141 | \[self: BasePlus =>
142 | 143 | def eql(other: exp): Boolean = other.isPlus(left, right);
144 | 145 | override def isPlus(l: exp, r: exp) = (left eql l) && (right eql r)
146 } final class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
147 148 | trait NegBehavior extends Exp {
149 | \[def eql(other: exp): Boolean = other.isNeg(term);
150 | 151 | override def isNeg(t: exp) = term eql t
152 } final class Neg(term: exp) extends BaseNeg(term) with NegBehavior with Exp
153 154 |
155 | trait EqualsShowPlusNeg extends EqualsPlusNeg with EqualsPlusNeg {
156 | type exp <% Exp
157 |
158 | trait Exp extends super[EqualsPlusNeg].Exp {
159 | \[def isPlus(l: exp, r: exp): Boolean = (left eql l) && (right eql r)
160 | 161 | trait PlusBehavior extends Exp {
162 | \[self: BasePlus =>
163 | 164 | class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
165 |
167 | \[self: BasePlus =>
168 | 169 } final class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
170 |
171 | trait NegBehavior extends super[EqualsPlusNeg].NegBehavior with super[ShowPlusNeg].NegBehavior {
172 | \[self: BaseNeg =>
173 } final class Neg(term: exp) extends BaseNeg(term) with NegBehavior with Exp
174 }--- 40 lines for equals
```

C.3.2 Isolated Scala

The following code is fully modularized: every trait defines exactly one operation for each datavariant.
trait Base {
  type exp <: Exp
}

trait Exp { }

trait BaseNum extends Base {
  class Num(val value: Int) extends Exp { }
  type BaseNum = Num
}

trait BasePlus extends Base {
  class Plus(val left: exp, val right: exp) extends Exp { }
  type BasePlus = Plus
}

trait BaseNeg extends Base {
  class Neg(val term: exp) extends Exp { }
  type BaseNeg = Neg
}

trait Eval extends Base {
  type exp <: Exp
  trait Exp { }
}

trait EvalNum extends BaseNum with Eval {
  trait NumBehavior {
    self: BaseNum =>
    def eval = value
  }
  class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
}

trait EvalPlus extends BasePlus with Eval {
  trait PlusBehavior {
    self: BasePlus =>
    def eval = left.eval + right.eval;
  }
  class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
}

trait EvalNeg extends BaseNeg with Eval {
  trait NegBehavior {
    self: BaseNeg =>
    def eval = - term.eval;
  }
  class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
}

trait Show extends Base {
  type exp <: Exp
  trait Exp { }
}

trait ShowNum extends BaseNum with Show {
  trait NumBehavior {
    self: BaseNum =>
    def show = value.toString
  }
  class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp
}

trait ShowPlus extends BasePlus with Show {
  trait PlusBehavior {
    self: BasePlus =>
    def show = left.show + " + " + right.show;
  }
  class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp
}

trait ShowNeg extends BaseNeg with Show {
  trait NegBehavior {
    self: BaseNeg =>
   }
Separating Use and Reuse to Improve Both

```scala
  def show = "-" + term.show + ""

class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp

trait Dble extends Base {
  type exp <: Exp
}

trait Dble extends Base {
  type exp <: Exp
}

trait Exp extends super[Exp] {
  def dble: exp
}

trait NumBehavior {
  self: BaseNum =>
  def dble = Num(value * 2)
}

def Num(v: Int): exp

class Num(v: Int) extends super[Num](v) with NumBehavior with Exp

trait DblePlus extends BasePlus with Dble {
  type exp <: Exp
}

trait PlusBehavior {
  self: BasePlus =>
  def dble = Plus(left.dble, right.dble)
}

def Plus(l: exp, r: exp): exp

class Plus(l: exp, r: exp) extends super[Plus](l, r) with PlusBehavior with Exp

trait DbleNeg extends BaseNeg with Dble {
  type exp <: Exp
}

trait NegBehavior {
  self: BaseNeg =>
  def dble = Neg(term.dble)
}

def Neg(t: exp): exp

class Neg(t: exp) extends super[Neg](t) with NegBehavior with Exp
```

---

```
object All0 extends Eval with Show with Dble with EvalNum with EvalPlus with EvalNeg with ShowNum with ShowPlus with ShowNeg with DbleNum with DblePlus with DbleNeg {
  override type exp = Exp


  trait NumBehavior extends super[EvalNum].NumBehavior with super[ShowNum].NumBehavior with super[DbleNum].NumBehavior {
    self: BaseNum =>
  }

  class Num(v: Int) extends super[Num](v) with NumBehavior with Exp

    self: BasePlus =>
  }

  class Plus(l: exp, r: exp) extends super[Plus](l, r) with PlusBehavior with Exp

  trait NegBehavior extends super[EvalNeg].NegBehavior with super[ShowNeg].NegBehavior with super[DbleNeg].NegBehavior {
```

---

Note: The text contains some typos and formatting issues, but it is readable and represents the code accurately.
class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp

def Num(v: Int) = new Num(v)
def Plus(l: exp, r: exp) = new Plus(l, r)
def Neg(t: exp) = new Neg(t)

trait Equals extends Base {
  type exp <: Exp;
}

trait Exp extends super[Base].Exp {
  def eql(other: exp): Boolean;
}

trait NumBehavior extends Exp {
  self: BaseNum =>
  def eql(other: exp): Boolean = other.isNum(value);
  override def isNum(v: Int) = v == value;
}

class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp

trait EqualsNum extends BaseNum with Equals {
  type exp <: Exp;
}

trait Exp extends super[BasePlus].Exp with super[Equals].Exp {
  def isPlus(l: exp, r: exp): Boolean;
}

trait PlusBehavior extends Exp {
  self: BasePlus =>
  def eql(other: exp): Boolean = other.isPlus(left, right);
  override def isPlus(l: exp, r: exp) = (left eql l) && (right eql r)
}

class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp

trait EqualsPlus extends BasePlus with Equals {
  type exp <: Exp;
}

trait Exp extends super[BaseNeg].Exp with super[Equals].Exp {
  def isNeg(t: exp): Boolean;
}

trait NegBehavior extends Exp {
  self: BaseNeg =>
  def eql(other: exp): Boolean = other.isNeg(term);
  override def isNeg(t: exp) = term eql t;
}

class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp

trait EqualsNeg extends BaseNeg with Equals {
  type exp <: Exp;
}

trait Exp extends super[BaseNeg].Exp with super[Equals].Exp {
  def isNeg(t: exp): Boolean;
}

trait NegBehavior extends Exp {
  self: BaseNeg =>
  def eql(other: exp): Boolean = other.isNeg(term);
  override def isNeg(t: exp) = term eql t;
}

class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp
Separating Use and Reuse to Improve Both

```scala
class Num(v: Int) extends BaseNum(v) with NumBehavior with Exp

  self: BasePlus =>
}

class Plus(l: exp, r: exp) extends BasePlus(l, r) with PlusBehavior with Exp

  self: BaseNeg =>
}

class Neg(t: exp) extends BaseNeg(t) with NegBehavior with Exp

def Num(v: Int) = new Num(v)
def Plus(l: exp, r: exp) = new Plus(l, r)
def Neg(t: exp) = new Neg(t)
```

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. In 42, these short cuts can be very expressive but here we use them only to better align the full 42 and 42 languages.

```scala
RenNum: {class method Library<< (Library that) exception Guard
  Refactor2.rename(path:"T" into:"Num")=>that
}

RenPlus: {class method Library<< (Library that) exception Guard
  Refactor2.rename(path:"T" into:"Plus")=>that
}

RenNeg: {class method Library<< (Library that) exception Guard
  Refactor2.rename(path:"T" into:"Neg")=>that
}

//above, header not counted

$exp:Resource<< { //3 lines
  Exp: {interface
    method Size value()
    class method Num of(Size value)
  }
}

$plus:Resource<< { //6 lines
  Exp: {interface
    method Exp left()
    method Exp right()
    class method Plus of(Exp left, Exp right)
  }
}

$neg:Resource<< { //5 lines
  Exp: {interface
    method Exp term()
    class method Neg of(Exp term)
  }
}

$eval:Resource<< { //4 lines
  Exp: {interface
    method Size eval()
  }
}

T: {implements Exp}
```
Use $evalNum:Resource<><$eval()
{ // 2*3 lines
  Num:
  method Size eval() this.value()
}

Use $evalPlus:Resource<><$eval()
{ // 2*3 lines
  Plus:
  method Size eval() this.left().eval() + this.right().eval()
}

Use $evalNeg:Resource<><$eval()
{ // 2*3 lines
  Neg:
  method Size eval() Size"-1" * this.term().eval()
}

Use $show:Resource<><$exp()
{ // 4 lines
  Exp:
  interface
  method S show()
  T:
    implements Exp
}

Use $showNum:Resource<><$num(); RenNum<><$show()
{ // 2*3 lines
  Num:
  method S show() this.value().toS()
}

Use $showPlus:Resource<><$plus(); RenPlus<><$show()
{ // 2*3 lines
  Plus:
  method S show() this.left().show() ++ S" + " ++ this.right().show()
}

Use $showNeg:Resource<><$neg(); RenNeg<><$show()
{ // 2*3 lines
  Neg:
  method S show() S"-" ++ this.term().show()
}

Use $double:Resource<><$double()
{ // 4 lines
  Exp:
  interface
  method Exp double()
  T:
    implements Exp
}

Use $doubleNum:Resource<><$num(); RenNum<><$double()
{ // 2*3 lines
  Num:
  method Exp double() Num.of(value: this.value() * 2)
}

Use $doublePlus:Resource<><$plus(); RenPlus<><$double()
{ // 2*3 lines
  Plus:
  method Exp double() Plus.of(left: this.left().double(), right: this.right().double())
}

Use $doubleNeg:Resource<><$neg(); RenNeg<><$double()
{ // 2*3 lines
  Neg:
  method Exp double() Neg.of(term: this.term().double())
}

Use $equals:Resource<><$eval()
{ // 6 lines
  Exp:
  interface
  method Bool eql(Exp that)
  method Bool equalToT(T that)
  T:
    implements Exp
}

Use $equalsNum:Resource<><$num(); $equals()
{ // 5*3 lines
  Refactor2.Methodrename:"equalToT(that)" into:"equalToNum(that)"
  Num:
  method Bool equalToNum(Num that) this.value() == that.value()
}

Use $equalsPlus:Resource<><$plus(); $equals()
{ // 5*3 lines
  Refactor2.Methodrename:"equalToT(that)" into:"equalToPlus(that)"
  Plus:
  method Bool equalToPlus(Plus that)
}

Use $equalsNeg:Resource<><$neg(); $equals()
{ // 5*3 lines
  Refactor2.Methodrename:"equalToT(that)" into:"equalToNeg(that)"
  Neg:
  method Bool equalToNeg(Neg that)
}

Use $equals:Resource<><$exp()
{ // 4 lines
  Exp:
    interface
  method Bool eq(T that)
}

Use $equals:Resource<><$exp()
{ // 4 lines
  Exp:
    interface
  method Bool eq(T that)
}

54 $evalNum:Resource<><$eval()
55 Num:
  method Size eval() this.value()
56 }
57 }
58 $evalPlus:Resource<><$eval()
59 Plus:
  method Size eval() this.left().eval() + this.right().eval()
60 }
61 $evalNeg:Resource<><$eval()
62 Neg:
  method Size eval() Size"-1" * this.term().eval()
63 }
64 }
65 $show:Resource<><$exp()
66 Exp:
  interface
  method S show()
67 }
68 }
69 }
70 $showNum:Resource<><$num(); RenNum<><$show()
71 Num:
  method S show() this.value().toS()
72 }
73 }
74 $showPlus:Resource<><$plus(); RenPlus<><$show()
75 Plus:
  method S show() this.left().show() ++ S" + " ++ this.right().show()
76 }
77 }
78 $showNeg:Resource<><$neg(); RenNeg<><$show()
79 Neg:
  method S show() S"-" ++ this.term().show()
80 }
81 }
82 $double:Resource<><$double()
83 Exp:
  interface
  method Exp double()
84 }
85 }
86 $doubleNum:Resource<><$num(); RenNum<><$double()
87 Num:
  method Exp double() Num.of(value: this.value() * 2)
88 }
89 }
90 $doublePlus:Resource<><$plus(); RenPlus<><$double()
91 Plus:
  method Exp double() Plus.of(left: this.left().double(), right: this.right().double())
92 }
93 }
94 $doubleNeg:Resource<><$neg(); RenNeg<><$double()
95 Neg:
  method Exp double() Neg.of(term: this.term().double())
96 }
97 $equals:Resource<><$eval()
98 Exp:
  interface
  method Bool eql(Exp that)
  method Bool equalToT(T that)
  T:
    implements Exp
99 }
100 $equalsNum:Resource<><$num(); $equals()
101 Num:
  method Bool equalToNum(Num that)
102 }
103 }
104 $equalsPlus:Resource<><$plus(); $equals()
105 Plus:
  method Bool equalToPlus(Plus that)
106 }
107 }
108 $equalsNeg:Resource<><$neg(); $equals()
109 Neg:
  method Bool equalToNeg(Neg that)
110 }
111 }
112 $equals:Resource<><$exp()
113 Exp:
  interface
  method Bool eq(T that)
114 method Bool eq(T that)
115 }
116 }
117 T:
  implements Exp
118 method eq(T that) that.equalToT(this)
119 }
120 }
121 }
122 $equalsNum:Resource<><$num(); $equals()
123 Num:
  method Bool equalToNum(Num that) this.value() == that.value()
124 }
125 }
126 $equalsPlus:Resource<><$plus(); $equals()
127 Plus:
  method Bool equalToPlus(Plus that)
128 }
129 }
130 $equalsNeg:Resource<><$neg(); $equals()
131 Neg:
  method Bool equalToNeg(Neg that) this.term().eval()
Separating Use and Reuse to Improve Both

```java
145 } //16+6 lines glue code
146 } 147 }
148 }
149 //16+6 lines glue code
150 }
151 }
152 }
153 }
154 }
155 }
156 }
157 }
158 }
159 }
160 }
161 }
162 }
163 }
164 }
165 }
166 }
167 }
168 }
169 }
170 }
171 }
172 }
173 }
174 }
175 }
176 }
177 }
178 }
179 }
180 }
181 }
182 }
183 }
184 }
185 }
186 }
187 }
188 }
189 }
190 }
191 }
192 }
193 }
194 }
195 }
196 }
197 }
198 }
199 }
200 }
201 }
202 }
203 }
204 }
205 }
206 }
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
```