Classless Java

Yanlin Wang Haoyuan Zhang
Bruno C. d. S. Oliveira
The University of Hong Kong, China
{ylwang,hyzhang,bruno}@cs.hku.hk

Marco Servetto
Victoria University of Wellington, New Zealand
marco.servetto@ecs.vuw.ac.nz

Abstract

This paper presents an OO style without classes, which we call interface-based object-oriented programming (IB). IB is a natural extension of closely related ideas such as traits. *Abstract state operations* provide a new way to deal with state, which allows for flexibility not available in class-based languages. In IB state can be type-refined in subtypes. The combination of a purely IB style and type-refinement enables powerful idioms using multiple inheritance and state. To introduce IB to programmers we created Classless Java: an embedding of IB directly into Java. Classless Java uses annotation processing for code generation and relies on new features of Java 8 for interfaces. The code generation techniques used in Classless Java have interesting properties, including guarantees that the generated code is type-safe and good integration with IDEs. Usefulness of IB and Classless Java is shown with examples and case studies.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Object-Oriented Programming; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

General Terms Languages

Keywords Interface-based programming, multiple inheritance, code generation

1. Introduction

Object-oriented languages strive to offer great code reuse. They couple flexibility and rigor, expressive power and modular reasoning. Two main OO models emerged to this end: prototype-based (PB) [22] and class-based languages such as Java, C# or Scala. In prototype-based languages objects inherit from other objects. Thus objects own both behavior and state (and objects are all you have). In class-based languages an object is an instance of a specific class, and classes inherit from other classes. Objects own state, while classes contain behavior and the structure of the state.

Regardless of the OO model, inheritance is a key mechanism is OO languages. Inheritance provides a modularization mechanism, which is used to reuse implementations from inherited classes/objects. Unfortunately, as widely acknowledged in the literature [5, 14, 19, 20] multiple inheritance, especially when combined with state, has several tricky issues, including several variants of the famous *diamond problem* [5, 19]. Many of the problems related to inheritance arise from the direct use of *fields* to model state. Inheriting two fields with the same name raises the question of whether the two fields should be kept, or only one field should exist. Initialization of the fields is also problematic, since initialization code may be inherited from multiple parents. Finally, an additional problem with mutable fields is that their type cannot be type-refined in extensions, which can cause modularity problems [23, 24].

To address those limitations, this paper presents a third alternative OO model called *interface-based* object-oriented programming languages (IB), where objects implement interfaces directly and fields are not directly supported. In IB interfaces own the implementation for the behavior, which is structurally defined in their interface. Programmers do not define objects directly but delegate the task to *object interfaces*, whose role is similar to non-abstract classes in class-based object-oriented programming languages (CB). Objects are instantiated by static factory methods in object interfaces.

Due to the absence of fields, a key challenge in IB lies in how to model state, which is fundamental to have stateful objects. All abstract operations in an object interface are interpreted as *abstract state operations*. The abstract state operations include various common utility methods (such as getters and setters, or clone-like methods). Objects are only responsible to define the ultimate behavior of a method. Anything related to state is completely contained in the instances and does not leak into the inheritance logic. In CB, the structure of the state is fixed and can only grow by inheritance. In contrast, in IB the state is never fixed, and methods such as abstract setters and getters can always receive an explicit implementation down in the inheritance

GPCE'16, October 31 – November 1, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4446-3/16/10...$15.00
http://dx.doi.org/10.1145/2993236.2993238

chain, improving **modularity and flexibility**. That is, the concept of abstract state is more fluid.

Object interfaces provide support for automatic type-refinement. In contrast, in CB special care and verbose explicit type-refinement are required to produce code that deals with subtyping adequately. We believe that such verbosity hindered and slowed down the discovery of useful programming patterns involving type-refinement. Our previous work [24] on the Expression Problem [23] in Java-like languages shows how easy it is to solve the problem using only type-refinement. However it took nearly 20 years since the formulation of the problem for that solution to be presented in the literature. In IB, due to its emphasis on type-refinement, that solution should have been more obvious.

One advantage of abstract state operations and type-refinement is that it allows a new approach to *type-safe covariant mutable state*. That is, in IB, it is possible to type-refine *mutable* state in subtypes. This is typically forbidden in CB: it is widely known that *naive* type-refinement of mutable fields is not type-safe. Although covariant refinement of mutable fields is supported by some type systems [7, 8, 10, 18], this requires significant complexity and restrictions to ensure that all uses of covariant state are indeed type-safe.

IB could be explained by defining a novel language, with new syntax and semantics. However, this would have a steep learning curve. We take a different approach instead. For the sake of providing a more accessible explanation, we will embed our ideas directly into Java. Our IB embedding relies on the new features of Java 8: interface *static methods* and *default methods*, which allow interfaces to have method implementations. In the context of Java, what we propose is a programming style, where we never use classes (more precisely, we never use the `class` keyword). We call this restricted version of Java *Classless Java*.

Using Java annotation processors, we produce an implementation of Classless Java, which allows us to stick to pure Java 8. The implementation works by performing AST rewriting, allowing most existing Java tools (such as IDEs) to work out-of-the-box with our implementation. Moreover, the implementation blends Java's conventional CB style and IB smoothly. We apply object interfaces to several interesting Java programs and conduct various case studies. Finally, we also discuss the behavior of Classless Java and its properties.

In summary, the contributions of this paper are:

- **IB and Object Interfaces:** which enable powerful programming idioms using multiple-inheritance, type-refinement and abstract state operations.
- **Classless Java:** a practical realization of IB in Java. Classless Java is implemented using annotation processing, allowing most tools to work transparently with our approach. Existing Java projects can use our approach and still be backward compatible with their clients, in a way that is specified by our safety properties.
- **Type-safe covariant mutable state:** we show how the combination of abstract state operations and type-refinement enables a form of mutable state that can be covariantly refined in a type-safe way.
- **Applications and case studies:** we illustrate the usefulness of IB through various examples and case studies[1]. An extended version with a formal translation to Java can be found in the companion technical report [25].

2. A Running Example: Animals

This section illustrates how our programming style, supported by `@Obj`, enables powerful programming idioms based on multiple inheritance and type refinements. We propose a standard example: `Animals` with a 2-dimensional `Point2D` representing their `location`, subtypes `Horses`, `Birds`, and `Pegasus`. Birds can `fly`, thus their locations need to be 3-dimensional `Point3Ds` (field type refinement). We model `Pegasus` (a well-known creature in Greek mythology) as a kind of `Animal` with the skills of both `Horses` and `Birds` (multiple inheritance). A simple class diagram illustrating the basic system is given on the left side of Figure 1.

2.1 Simple Multiple Inheritance with Default Methods

Before modelling the complete animal system, we start with a simple version without locations. This version serves the purpose of illustrating how Java 8 default methods can already model simple forms of multiple inheritance. `Horse` and `Bird` are subtypes of `Animal`, with methods `run()` and `fly()`, respectively. Pegasus can not only *run* but also *fly*! This is the place where *"multiple inheritance"* is needed because `Pegasus` needs to obtain `fly` and `run` functionality from both `Horse` and `Bird`. A first attempt to model the animal system is given on the right side of Figure 1. Note that the implementations of the methods `run` and `fly` are defined inside interfaces, using default methods. Moreover, because interfaces support multiple interface inheritance, the interface for `Pegasus` can inherit behavior from both `Horse` and `Bird`. Although Java interfaces do not allow instance fields, no form of state is needed so far to model the animal system.

Instantiation To use `Horse`, `Bird` and `Pegasus`, some objects must be created first. A first problem with using interfaces to model the animal system is simply that interfaces cannot be directly instantiated. Classes, such as:

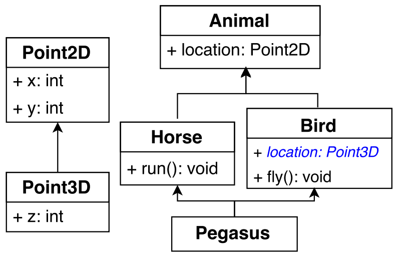
```
class HorseImpl implements Horse {}
class BirdImpl implements Bird {}
class PegasusImpl implements Pegasus {}
```

are needed for instantiation. Now a `Pegasus` animal can be created using the class constructor:

```
Pegasus p = new PegasusImpl();
```

There are some annoyances here. Firstly, the sole purpose of the classes is to provide a way to instantiate objects. Although (in this case) it takes only one line of code to provide each of those classes, this code is essentially boilerplate code, which

[1] `https://github.com/YanlinWang/classless-java`



```java
interface Animal {} // no points yet!
interface Horse extends Animal {
   default void run(){out.println("run!");}
}
interface Bird extends Animal {
   default void fly(){out.println("fly!");}
}
interface Pegasus extends Horse, Bird {}
```

Figure 1. The animal system (left: complete structure, right: code for simplified animal system).

does not add behavior to the system. Secondly, the namespace gets filled with three additional types. For example, both `Horse` and `HorseImpl` are needed: `Horse` is needed because it needs to be an interface so that `Pegasus` can use multiple inheritance; and `HorseImpl` is needed to provide object instantiation. Note that, for this very simple animal system, plain Java 8 anonymous classes can be used to avoid these problems. We could have simply instantiated `Pegasus` using:

```java
Pegasus p = new Pegasus() {}; // anonymous class
```

However, as we shall see, once the system gets a little more complicated, the code for instantiation quickly becomes more complex and verbose (even with anonymous classes).

2.2 Object Interfaces and Instantiation

To model the animal system with object interfaces all that a user needs to do is to add an `@Obj` annotation to the `Horse`, `Bird`, and `Pegasus` interfaces:

```java
@Obj interface Horse extends Animal {
  default void run() {out.println("running!");} }
@Obj interface Bird extends Animal {
  default void fly() {out.println("flying!");} }
@Obj interface Pegasus extends Horse, Bird {}
```

The effect of the annotations is that a static *factory* method called `of` is automatically added to the interfaces. With the `of` method a `Pegasus` object is instantiated as follows:

```java
Pegasus p = Pegasus.of();
```

The `of` method provides an alternative to a constructor, which is missing from interfaces. The following code shows the code corresponding to the `Pegasus` interface after the `@Obj` annotation is processed:

```java
interface Pegasus extends Horse, Bird {
  // generated code not visible to users
  static Pegasus of() { return new Pegasus() {}; } }
```

Note that the generated code is transparent to users, who only see the original code with the `@Obj` annotation. Compared to the pure Java solution in Section 2.1, the solution using object interfaces has the advantage of providing a direct mechanism for object instantiation, which avoids adding boilerplate classes to the namespace.

2.3 Object Interfaces with State

The animal system modeled so far is a simplified version of the system presented in the left-side of Figure 1. The example is still not sufficient to appreciate the advantages of IB programming. Now we model the complete animal system where an `Animal` includes a `location` representing its position in space. We use 2D points to keep track of locations.

***`Point2D`*: simple immutable data with fields** Points can be modeled with interfaces. In IB state is accessed and manipulated using abstract methods. The usual approach to model points in Java is to use a class with fields for the coordinates. In Classless Java interfaces are used instead:

```java
interface Point2D { int x(); int y(); }
```

The encoding over Java is now inconvenient: creating a new point object is cumbersome, even with anonymous classes:

```java
Point2D p = new Point2D() {
  public int x() {return 4;}
  public int y() {return 2;} }
```

However this cumbersome syntax is not required for every object allocation. As programmers do, for ease or reuse, the boring repetitive code can be encapsulated in a method. A generalization of the `of` static factory method is appropriate:

```java
interface Point2D { int x(); int y();
  static Point2D of(int x, int y) {
    return new Point2D() {
      public int x(){return x;}
      public int y(){return y;}
    }; } }
```

***`Point2D`* with object interfaces** This obvious "constructor" code is generated by the `@Obj` annotation. By annotating the interface `Point2D`, a variation of the shown static method `of` will be generated, mimicking the functionality of a simple-minded constructor. `@Obj` first looks at the abstract methods and detects what the fields are, then generates an `of` method with one parameter for each of them. We can just write:

```java
@Obj interface Point2D { int x(); int y(); }
```

A field or factory parameter is generated for every abstract method that takes no parameters. An example of using `Point2D`, where we "clone" an existing point but use `42` as the x-coordinate, is:

```java
Point2D p = Point2D.of(42,myPoint.y());
```

***`with-`* methods in object interfaces** The pattern of creating a new object by reusing most information from an old object is very common when programming with immutable data-structures. As such, it is supported by `@Obj` as `with-` methods:

```java
@Obj interface Point2D {
   int x(); int y(); // getters
   // with- methods
   Point2D withX(int val);
   Point2D withY(int val); }
```

Using `with-` methods, the point `p` can also be created by:

```java
Point2D p = myPoint.withX(42);
```

If there is a large number of fields, `with-` methods will save programmers from writing large amounts of tedious code that simply copies field values. Moreover, if the programmer wants a different implementation, he may provide an alternative implementation using **`default`** methods. For example:

```
@Obj interface Point2D {
   int x(); int y();
   default Point2D withX(int val){ /*myCode*/ }
   Point2D withY(int val); }
```

is expanded into

```
interface Point2D {
   int x(); int y();
   default Point2D withX(int val){ /*myCode*/ }
   Point2D withY(int val);
   static Point2D of(int _x, int _y){
     return new Point2D(){
       int x=_x; int y=_y;
       public int x(){return x;}
       public int y(){return y;}
       public Point2D withY(int val){
         return of(x(),val);} }; } }
```

Only code for methods needing implementation is generated. Thus, programmers can easily customize the behavior for their special needs. Also, since `@Obj` interfaces offer the `of` factory method, only interfaces where all the abstract methods can be synthesized can be object interfaces. A non-`@Obj` interface is like an abstract class in Java.

`Animal` and `Horse`: simple mutable data with fields 2D points are mathematical entities, thus we choose an immutable data structure to model them. Animals are real world entities, and when an animal moves, it is the *same* animal with a different location. We model this with mutable state.

```
interface Animal {
  Point2D location();
  void location(Point2D val); }
```

Here we declare an abstract getter and a setter for the mutable "field" `location`. Without the `@Obj` annotation, there is no convenient way to instantiate `Animal`. For `Horse`, the `@Obj` annotation is used and an implementation of `run()` is defined using a **`default`** method. The implementation of `run()` further illustrates the convenience of `with-` methods:

```
@Obj interface Horse extends Animal {
  default void run() {
    location(location().withX(location().x()+20));}}
```

Creating and using `Horse` is quite simple:

```
Point2D p = Point2D.of(0, 0);
Horse horse = Horse.of(p);
horse.location(p.withX(42));
```

Note how the `of`, `withX` and `location` methods (generated automatically) give a basic interface for dealing with animals.

In summary, state (mutable or not) in object interfaces relies on a notion of abstract state, and state is not directly available to programmers. Instead programmers use methods, called *abstract state operations*, to interact with state.

2.4 Object Interfaces and Subtyping

`Birds` are `Animals`, but while `Animals` only need 2D locations, `Birds` need 3D locations. Therefore when the `Bird` interface extends the `Animal` interface, the notion of points needs to be *refined*. Such kind of refinement is challenging in typical class-based approaches. Fortunately, with object interfaces, we are able to provide a simple and effective solution.

Unsatisfactory class-based solutions to field type refinement In Java if we want to define an animal class with a field we have a set of unsatisfactory options in front of us:

- Define a `Point3D` field in `Animal`: this is bad since all animals would require more than needed. Also it requires adapting the old code to accommodate for new evolutions.
- Define a `Point2D` field in `Animal` and define an extra **`int`** `z` field in `Bird`. This solution is very ad-hoc, requiring to basically duplicate the difference between `Point2D` and `Point3D` inside `Bird`. The most dramatic criticism is that it would not scale to a scenario when `Bird` and `Point3D` are from different programmers.
- Redefine getters and setters in `Bird`, always put `Point3D` objects in the field and cast the value out of the `Point2D` field to `Point3D` when implementing the overridden getter. This solution scales to the multiple programmers approach, but requires ugly casts and can be implemented in a wrong way leading to bugs.

We may be tempted to assume that a language extension is needed. Instead, the *restriction* of (object) interfaces to have no fields enlightens us that another approach is possible; often in programming languages "freedom is slavery".

Field type refinement with object interfaces Object interfaces address the challenge of type-refinement as follows:

- by *covariant method overriding*, the return type of `location()` is refined to `Point3D`;
- by *overloading*, a new setter for location is defined with a more precise type;
- a **`default`** setter implementation with the old signature is provided.

Thus the code for the `Bird` interface is:

```
@Obj interface Bird extends Animal {
  Point3D location(); void location(Point3D val);
  default void location(Point2D val) {
    location(location().with(val));}
  default void fly() {
    location(location().withX(
      location().x() + 40));} }
```

From the type perspective, the key is the covariant method overriding of `location()`. However, from the semantics perspective the key is the implementation for the setter with the old signature (`location(Point2D)`). The key to the setter implementation is a new type of `with` method, called a (functional) property updater.

`Point3D` and property updaters The `Point3D` interface is defined as follows:

```
@Obj interface Point3D extends Point2D {
   int z();
   Point3D withZ(int z);
   Point3D with(Point2D val); }
```

`Point3D` includes a `with` method, taking a `Point2D` as an argument. Other wither methods (such as `withX`) functionally

```
interface Point3D extends Point2D {
  int z(); Point3D withZ(int val);
  Point3D with(Point2D val);
  // generated code
  Point3D withX(int val);
  Point3D withY(int val);
  public static Point3D of(int _x, int _y, int _z){
    int x=_x; int y=_y; int z=_z;
    return new Point3D(){
      public int x(){return x;}
      public int y(){return y;}
      public int z(){return z;}
      public Point3D withX(int val){
        return Point3D.of(val, this.y(), this.z());
      }
      public Point3D withY(int val){
        return Point3D.of(this.x(), val, this.z());
      }
      public Point3D withZ(int val){
        return Point3D.of(this.x(), this.y(), val);
      }
      public Point3D with(Point2D val){
        if(val instanceof Point3D)
          return (Point3D)val;
        return Point3D.of(val.x(), val.y(), this.z());
      }
    }; } }
```

Figure 2. Generated boilerplate code.

update a field one at a time. This can be inefficient, and sometimes hard to maintain. Often we want to update multiple fields simultaneously, for example using another object as source. Following this idea, the method `with(Point2D)` is an example of a (functional) property updater: it takes a certain type of object and returns a copy of the current object where all the fields that match fields in the parameter object are updated to the corresponding value in the parameter. The idea is that the result should be like **`this`**, but modified to be as similar as possible to the parameter.

With the new `with` method we may use the information for `z` already stored in the object to forge an appropriate `Point3D` to store. Note how all the information about what fields sit in `Point3D` and `Point2D` is properly encapsulated in the `with` method, and is transparent to the implementer of `Bird`.

Property updaters never break class invariants, since they internally call operations that were already deemed safe by the programmer. For example a list object would not offer a setter for its `size` field (which should be kept hidden), thus a property updater would not attempt to set it.

Generated boilerplate To give an idea of how much code `@Obj` is generating, we show the generated code for `Point3D` in Figure 2. Writing such code by hand is error-prone. For example a distracted programmer may swap the arguments of calls to `Point3D.of`. Note how `with-` methods are automatically refined in their return types, so that code like:

```
Point3D p = Point3D.of(1,2,3); p = p.withX(42);
```

will be accepted. If the programmer wishes to suppress this behavior and keep the signature as it was, it is sufficient to redefine the `with-` methods in the new interface repeating the old signature. Again, the philosophy is that if the programmer provides something directly, `@Obj` does not touch it. The cast in `with(Point2D)` is trivially safe because of the **`instanceof`** test. The idea is that if the parameter is a subtype of the current exact type, then we can just return the parameter, as something that is just "more" than **`this`**.

Summary of operations in Classless Java In summary, object interfaces provide support for different types of abstract state operations: four field-based state operations; and functional updaters. Object instantiation is directly supported by `of` factory methods. Figure 3 summarizes the six operations supported by `@Obj`. The field-based abstract state operations are determined by naming conventions and the types of the methods. Fluent setters are a variant of conventional setters, and are discussed in more detail in Section 4.2.

2.5 Advanced Multiple Inheritance

Finally, defining `Pegasus` is as simple as we did in the simplified (and stateless) version on the right of Figure 1. Note how even the non-trivial pattern for field type refinement is transparently composed, and `Pegasus` has a `Point3D location`.

```
@Obj interface Pegasus extends Horse, Bird {}
```

3. Bridging between IB and CB in Java

Creating a new language/extension would be an elegant way to illustrate the point of IB. However, significant amounts of engineering would be needed to build a practical language and achieve a similar level of integration and tool support as Java. To be practical, we have instead implemented `@Obj` as an annotation in Java 8, and a *compilation agent*. That is, the Classless Java style of programming is supported by library.

Disciplined use of Classless Java (avoiding class declarations as done in Section 2) illustrates what *pure* IB is like. However, using `@Obj`, CB and IB programming can be mixed together, harvesting the practical convenience of using existing Java libraries, the full Java language and IDE support. The key to our implementation is compilation agents, which allow us to rewrite the Java AST just before compilation. We discuss the advantages and limitations of our approach.

3.1 Compilation Agents

Java supports compilation agents, where Java libraries can interact with the Java compilation process, acting as a man in the middle between the generation of AST and bytecode.

This process is facilitated by frameworks like Lombok [29]: a Java library that aims at reducing Java boilerplate code via annotations. `@Obj` was created using Lombok. Figure 4 [15] illustrates the flow of the `@Obj` annotation. First Java source code is parsed into an abstract syntax tree (AST). The AST is then captured by Lombok: each annotated node is passed to the corresponding (Eclipse or Javac) handler. The handler is free to modify the information of the annotated node, or even inject new nodes (like methods, inner classes, etc). Finally, the Java compiler works on the modified AST to generate bytecode.

	Operation	Example	Description
State operations (for a field x)	**"fields"/getters**	`int x()`	Retrieves value from field x.
	withers	`Point2D withX(int val)`	Clones object; updates field x to val.
	setters	`void x(int val)`	Sets the field x to a new value val.
	fluent setters	`Point2D x(int val)`	Sets the field x to val and returns **this**.
Other operations	**factory methods**	`static Point2D of(int _x,int _y)`	Factory method (generated).
	functional updaters	`Point3D with(Point2D val)`	Updates all matching fields in val.

Figure 3. Abstract state operations for a field x, together with other operations, supported by the @Obj annotation.

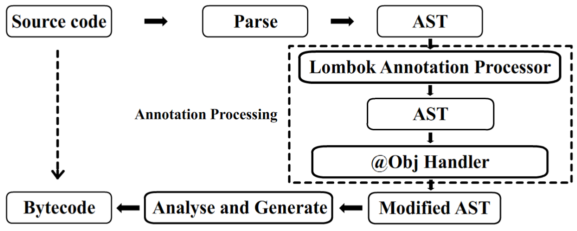


Figure 4. The flow chart of @Obj annotation processing.

Advantages of Lombok The Lombok compilation agent has advantages with respect to alternatives like pre-processors, or other Java annotation processors. Lombok offers in Java an expressive power similar to that of Scala/Lisp macros; except, for the syntactic convenience of quote/unquote templating.

Direct modification of the AST Lombok alters the generation process of the class files, by directly modifying the AST. Neither the source code is modified nor new Java files are generated. Moreover, and probably more importantly, Lombok supports generation of code *inside* a class/interface, which conventional Java annotation processors, such as `javax.annotation`, do not support.

Modularity While general preprocessing acts across module boundaries, compilation agents act modularly on each class/compilation unit. It makes sense to apply the transformations to one class/interface at a time, and only to annotated classes/interfaces. This allows library code to be reused without being reprocessed or recompiled, making our approach 100% compatible with existing Java libraries, which can be used and extended normally.

Tool support Features written in Lombok integrate and are supported directly in the language and are also supported by most tools. In Figure 5, `@Obj` generates an `of` method in `Point2D`, and `of`, `withX`, `withY` methods in `Point3D`. In Eclipse, the processing is performed transparently and the information of the interface from compilation is captured in the "Outline" window. This includes all the methods inside the interface, including the generated ones. Moreover, as a useful IDE feature, the auto-completion also works for these newly generated methods.

3.2 `@Obj` AST Reinterpretation

Of course, careless reinterpretation of the AST could still be surprising for badly designed rewritings. `@Obj` reinterprets the syntax with the sole goal of *enhancing and completing code*: we satisfy the behavior of abstract methods; add method implementations; and refine return types. We consider this to be quite easy to follow and reason about, since it is similar to what happens in normal inheritance. Refactoring operations like renaming and moving should work transparently in conjunction with our annotation, since they rely on the overall type structure of the class, which we do not arbitrarily modify but just complete.

Thus, in addition to the advantages of Lombok, Classless Java offers more advantages with respect to arbitrary (compilation agent driven) AST rewriting.

Syntax and type errors Some preprocessors (like the C one) can produce syntactically invalid code. Lombok ensures only syntactically valid code is produced. Classless Java additionally guarantees that no type errors are introduced in generated code and client code. We discuss these two guarantees in more detail next:

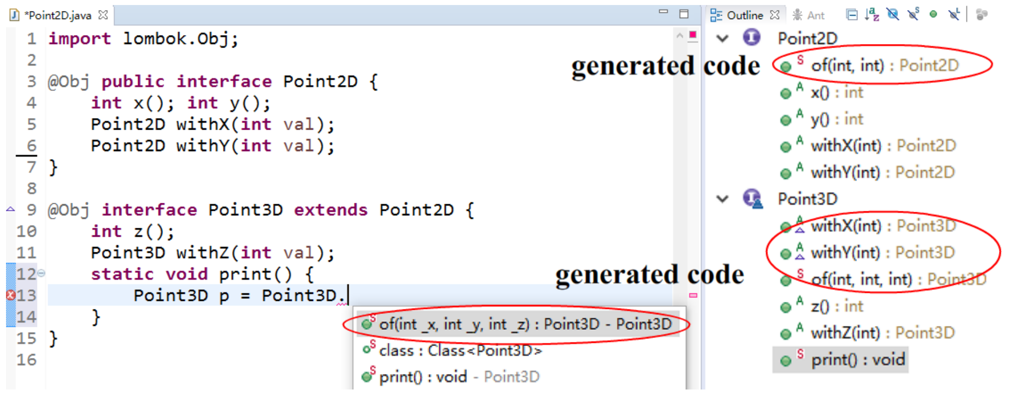
- **Self coherence**: the generated code itself is well-typed. In our case, it means that either `@Obj` produces (in a controlled way) an understandable error or the interface can be successfully annotated and the generated code (e.g. the `of` methods in Figure 5) is well-typed.
- **Client coherence**: all the client code (for example method calls) that is well-typed before code generation is also well-typed after the generation. The annotation just adds more behavior without removing any functionality.

Heir coherence Another form of guarantee that could be useful in AST rewriting is heir coherence. That is, interfaces (and in general classes) inheriting the instrumented code are well-typed if they were well-typed without the instrumentation. In a strict sense, our rewriting *does not* guarantee heir coherence. The reason is that this would forbid adding any (default or abstract) method to the annotated interfaces, or even doing type refinement. Indeed consider the following:

```
interface A { int x(); A withX(int x); }
@Obj interface B extends A {}
interface C extends B { A withX(int x); }
```

This code is correct before the translation, but `@Obj` would generate in `B` a method "`B withX(int x);`". This would break `C`. Similarly, an expression of the form "`new B(){.. A withX (int x){..}}`" would be correct before translation, but ill-typed after the translation.

Our automatic type refinement is a useful and convenient feature, but not transparent to the heirs of the annotated interface. They need to be aware of the annotation semantics and provide the right type while refining methods. To support heir coherence, we need to give up automatic type refinement,



```
import lombok.Obj;

@Obj public interface Point2D {
    int x(); int y();
    Point2D withX(int val);
    Point2D withY(int val);
}

@Obj interface Point3D extends Point2D {
    int z();
    Point3D withZ(int val);
    static void print() {
        Point3D p = Point3D.
    }
}
```

```
of(int _x, int _y, int _z) : Point3D - Point3D
class : Class<Point3D>
print() : void - Point3D
```

```
Point2D
  of(int, int) : Point2D
  x() : int
  y() : int
  withX(int) : Point2D
  withY(int) : Point2D
Point3D
  withX(int) : Point3D
  withY(int) : Point3D
  of(int, int, int) : Point3D
  z() : int
  withZ(int) : Point3D
  print() : void
```

Figure 5. Generated methods shown in the Outline window of Eclipse and auto-completion.

which is an essential part of IB programming. However, Java libraries almost always break heir coherence during evolution and still claim backward compatibility. In practice, adding any method to any non-final class of a Java library is enough to break heir coherence. We think return type refinement breaks heir coherence “less" than normal library evolution, and if no automatic type-refinements are needed, then `@Obj` can claim a form of heir coherence. Formal definition/proofs for our safety claims are in the technical report.

3.3 Limitations

Our prototype implementation has certain limitations:

- Lombok allows writing handlers for either javac or ejc(Eclipse’s own compiler). Our current implementation only realizes ejc version. The implementation for the `javac` version is still missing.
- Simple generics is supported: type parameters can be used, but generic method typing is delegated to the Java compiler instead of being explicitly checked by `@Obj`.
- Due to limited support in Lombok for separate compilation, i.e., accessing information of code defined in different files, `@Obj` requires that all related interfaces have to appear in a single Java file. Reusing the logic inside the experimental Lombok annotation `@Delegate`, we also offer a less polished annotation supporting separate compilation.

4. Applications and Case Studies

This section illustrates applications and larger case studies for Classless Java. The first application shows how a useful pattern, using multiple inheritance and type-refinement, can be conveniently encoded in Classless Java. The second application shows how to model embedded DSLs based on fluent APIs. Then two larger case studies refactor existing projects into Classless Java. The first one shows a significant reduction in code size, while the second one maintains the same amount of code, but improves modularity.

4.1 The Expression Problem with Object Interfaces

As the first application for Classless Java, we illustrate a useful programming pattern that improves modularity and extensibility of programs. This useful pattern is based on an existing solution to the *Expression Problem* (EP) [23], which is a well-known problem about modular extensibility issues in software evolution. Recently, a new solution [24] using only covariant type refinement was proposed. When this solution is modeled with interfaces and default methods, it can even provide independent extensibility [27]: the ability to assemble a system from multiple, independently developed extensions. Unfortunately, the required instantiation code makes a plain Java solution verbose and cumbersome to use. The `@Obj` annotation is enough to remove the boilerplate code, making the presented approach very appealing. Our last case study, presented in Section 4.4, is essentially a (much larger) application of this pattern to an existing program. Here we illustrate the pattern in the much smaller Expression Problem.

Initial system In the formulation of the EP, there is an initial system that models arithmetic expressions with only literals, addition, and an initial operation `eval` for expression evaluation. As shown in Figure 6, `Exp` is the common super-interface with operation `eval()` inside. Sub-interfaces `Lit` and `Add` extend interface `Exp` with default implementations for the `eval` operation. The number field `x` of a literal is represented as a getter method `x()` and expression fields (`e1` and `e2`) of an addition as getter methods `e1()` and `e2()`.

Adding a new type of expressions In the OO paradigm, it is easy to add new types of expressions. For example, the following code shows how to add subtraction.

```
@Obj interface Sub extends Exp {
  Exp e1(); Exp e2();
  default int eval() {
    return e1().eval() - e2().eval();} }
```

Adding a new operation The difficulty of the EP in OO languages arises from adding new operations. For example, adding a pretty printing operation would typically change all existing code. However, a solution should add operations in a type-safe and modular way. This turns out to be easily achieved with the assistance of `@Obj`. The code in Figure 6 (on the right) shows how to add the new operation `print`. Interface `ExpP` extends `Exp` with the extra method `print()`. Interfaces `LitP` and `AddP` are defined with default implementations of `print()`, extending base interfaces `Lit` and `Add`, respectively. Importantly, note that in `AddP`, the types

```
interface Exp { int eval(); }
@Obj interface Lit extends Exp {
   int x();
   default int eval() {return x();}
}
@Obj interface Add extends Exp {
   Exp e1(); Exp e2();
   default int eval() {
      return e1().eval() + e2().eval();
   }
}
```

```
interface ExpP extends Exp {String print();}
@Obj interface LitP extends Lit, ExpP {
   default String print() {return "" + x();}
}
@Obj interface AddP extends Add, ExpP {
   ExpP e1(); //return type refined!
   ExpP e2(); //return type refined!
   default String print() {
      return "(" + e1().print() + " + "
            + e2().print() + ")";}
}
```

Figure 6. The Expression Problem (left: initial system, right: code for adding print operation).

of "*fields*" (i.e. the getter methods) `e1` and `e2` are refined. If the types were not refined then the `print()` method in `AddP` would fail to type-check.

Independent extensibility To show that our approach supports independent extensibility, a new operation `collectLit` which collects all literal components in an expression is defined. For space reasons, we omit some code:

```
interface ExpC extends Exp {
   List<Integer> collectLit(); }
@Obj interface LitC extends Lit, ExpC {...}
@Obj interface AddC extends Add, ExpC {
   ExpC e1(); ExpC e2(); ...}
```

Now we combine the two extensions together:

```
interface ExpPC extends ExpP, ExpC {}
@Obj interface LitPC extends ExpPC, LitP, LitC {}
@Obj interface AddPC extends ExpPC, AddP, AddC {
    ExpPC e1(); ExpPC e2(); }
```

`ExpPC` is the new expression interface supporting `print` and `collectLit` operations; `LitPC` and `AddPC` are the extended variants. Notice that except for the routine of **extends** clauses, no glue code is required. Return types of `e1,e2` must be refined to `ExpPC`. Creating a simple expression of type `ExpPC` is as simple as:

```
ExpPC e8 = AddPC.of(LitPC.of(3), LitPC.of(4));
```

Without Classless Java, tedious instantiation code would need to be defined manually.

4.2 Embedded DSLs with Fluent Interfaces

Since the style of fluent interfaces was invented in Smalltalk as method cascading, more and more languages (Java, C++, Scala, etc) came to support fluent interfaces. In most languages, to create fluent interfaces, programmers have to either hand-write everything or create a wrapper around the original non-fluent interfaces using **this**. In Java, there are several libraries (including jOOQ, op4j, fluflu, JaQue, etc) providing useful fluent APIs. However most of them only provide a fixed set of predefined fluent interfaces.

The `@Obj` annotation can also be used to create fluent interfaces. When creating fluent interfaces with `@Obj`, there are two main advantages:

1. Instead of forcing programmers to hand-write code using **return this**, our approach with `@Obj` annotation removes this verbosity and automatically generates fluent setters.
2. The approach supports extensibility: the return types of fluent setters are automatically refined.

We use embedded DSLs of two simple SQL query languages to illustrate. The first query language `Database` models `select`, `from` and `where` clauses:

```
@Obj interface Database {
   String select(); Database select(String select);
   String from(); Database from(String from);
   String where(); Database where(String where);
   static Database of() {return of("", "", "");} }
```

The main benefit that fluent methods give us is the convenience of method chaining:

```
Database query1 = Database.of().select("a, b").from(
    "Table").where("c > 10");
```

Note how all the logic for the fluent setters is automatically provided by the `@Obj` annotation.

Extending the query language The previous query language can be extended with a new feature `orderBy` which orders the result records by a field that users specify. With `@Obj` programmers just need to extend the interface `Database` with new features, and the return type of fluent setters in `Database` is automatically refined to `ExtendedDatabase`:

```
@Obj interface ExtendedDatabase extends Database {
  String orderBy();
  ExtendedDatabase orderBy(String orderBy);
  static ExtendedDatabase of() {
    return of("", "", "","");} }
```

In this way, when a query is created using `ExtendedDatabase`, all the fluent setters return the correct type instead of the old `Database` type, which would prevent calling `orderBy`.

```
ExtendedDatabase query2 = ExtendedDatabase.of().
    select("a, b").from("Table").where("c > 10").
    orderBy("b");
```

Languages like Smalltalk and Dart offer method cascading and avoid the need for fluent setters. This is achieved at the price of introducing additional syntax and intrinsically relies on an imperative setting. Our approach supports both fluent setters and (functional) fluent withers.

4.3 A Maze Game

This case study is a simplified variant of a Maze game, which is often used [4, 11] to evaluate code reuse ability related to inheritance and design patterns. In the game, there is a player with the goal of collecting as many coins as possible. She may enter a room with several doors to be chosen among. This is a good example because it involves code reuse (different kinds of doors inherit a common type, with different features and behavior), multiple inheritance (a special kind of door may require features from two other door types) and it also shows how to model operations `symmetric sum`, `override` and `alias` from trait-oriented programming. The game has been implemented using plain Java 8 and default methods by Bono et. al [4], and the code for that implementation is available online. We refactored the game using `@Obj`. Due to space constraints, we omit the code here. Table 1 summarizes the number of lines of code and classes/interfaces in each implementation. The `@Obj` annotation reduced the interfaces/classes used in Bono et al.'s implementation by 21.4% (from 14 to 11), due to the replacement of instantiation classes with generated `of` methods. The number of source lines of code (SLOC) was reduced by 40% due to both the removal of instantiation overhead and generation of getters/setters.

	SLOC	# of classes/interfaces
Bono et al.	335	14
Ours	199	11
Reduced by	40.6%	21.4%

Table 1. Maze game code size comparison.

4.4 Refactoring an Interpreter

The last case study refactors the code from an interpreter for a Lisp-like language `Mumbler`[2], which is created as a tutorial for the Truffle Framework [26]. Keeping a balance between simplicity and useful features, `Mumbler` contains numbers, booleans, lists (encoding function calls and special forms such as if-expression, lambdas, etc). In the original code base, which consists of 626 SLOC of Java, only one operation `eval` is supported. Extending Mumbler to support one more operation, such as a pretty printer `print`, would normally require changing the existing code base directly.

Our refactoring applies the pattern presented in Section 4.1 to the existing Mumbler code base to improve its modularity and extensibility. Using the refactored code base it becomes possible to add new operations modularly, and to support independent extensibility. We add one more operation `print` to both the original and the refactored code base. In the original code base the pretty printer is added non-modularly by modifying the existing code. As shown in table 2 the pretty printer in the refactored code is added modularly with more interfaces, thus the code is slightly increased by 2.4% SLOC. However, the modularity is greatly increased, allowing for improved reusability and maintainability.

Code	SLOC	Code	SLOC
original (`eval`)	626	original (`eval+print`)	661
refactored (`eval`)	560	refactored (`eval+print`)	677

Table 2. Interpreter code size comparison.

[2] `https://github.com/cesquivias/mumbler/tree/master/simple`

5. Related Work

Multiple inheritance in object-oriented languages Many authors have argued in favor or against multiple inheritance. Multiple *implementation* inheritance is very expressive, but difficult to model and implement. It can cause difficulties (including the famous diamond problem [5, 19], conflicting methods, etc.) in reasoning about programs. To allow for expressive power and simplicity, many models have been proposed, including C++ virtual inheritance, mixins [5], traits [20], and hybrid models such as CZ [14]. They provide novel programming architecture models in the OO paradigm. In terms of restrictions set on these models, C++ virtual inheritance aims at a relative general model; the mixin model adds some restrictions; and the trait model is the most restricted one (excluding state, instantiation, etc).

C++ has a general solution to multiple inheritance by virtual inheritance, dealing with the diamond problem by keeping only one copy of the base class [9]. However it suffers from object initialization problem [14]. It bypasses constructor calls to virtual superclasses, which can cause serious semantic errors. In our approach, the `@Obj` annotation has full control over object initialization, and the mechanism is transparent to users. If users are not satisfied with the default `of` method, customized factory methods can be provided.

Mixins are more restricted than the C++ approach. Mixins allow naming components that can be applied to various classes as reusable functionality units. However, the linearization (total ordering) of mixin inheritance cannot provide a satisfactory resolution in some cases and restricts the flexibility of mixin composition. To fight against this limitation, an algebra of mixin operators is introduced [2], but this raises the complexity, especially when constructors and fields are considered [28]. Scala traits [16] are in fact more like linearized mixins. Scala avoids the object initialization problem by disallowing constructor parameters, causing no ambiguity in cases such as the diamond problem. However this approach has limited expressiveness, and suffers from all the problems of linearized mixin composition. Java interfaces and default methods do not use linearization: the semantics of Java **extends** clause in interfaces is unordered and symmetric.

Malayeri and Aldrich proposed a model CZ [14] which aims to do multiple inheritance without the diamond problem. Inheritance is divided into two concepts: inheritance dependency and implementation inheritance. Using a combination of `requires` and `extends`, a program with diamond inheritance is transformed to one without diamonds. Moreover, fields and multiple inheritance can coexist. However untangling inheritance also untangles the class structure. In CZ, not only the number of classes, but also the class hierarchy complexity increases. IB does not complicate the hierarchical structure, and state also coexists with multiple inheritance.

Simplifying the mixins approach, traits [20] draw a strong line between units of reuse and object factories. Traits, as units of reusable code, contain only methods as reusable functionality, ignoring state and state initialization. Classes, as object factories, require functionality from (multiple) traits. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the **default** keyword) inside interfaces. The introduction of default methods opens the gate for various flavors of multiple inheritance in Java. Traits offer an algebra of composition operations like sum, alias, and exclusion, providing explicit conflict resolution. Former work [4] provides details on mimicking the trait algebra through Java 8 interfaces. There are also proposals for extending Java with traits. For example, FeatherTrait Java (FTJ) [13] extends FJ [12] with statically-typed traits, adding trait-based inheritance in Java. Except for few, mostly syntactic details, their work can be emulated with Java 8 interfaces. There are also extensions to the original trait model, with operations (e.g. renaming [17], which breaks structural subtyping) that default methods and interfaces cannot model.

Traits vs object interfaces. We consider object interfaces an alternative to traits or mixins. In trait model two concepts (traits and classes) coexist and cooperate. Some authors [3] see this as good language design fostering good software development by helping programmers think about the program structures. However, others see the need of two concepts and the absence of state as drawbacks of this model [14]. Object interfaces are units of reuse, and meanwhile provide factory methods for instantiation and support state. Our approach promotes the use of interfaces in order to exploit the modular composition offered by interfaces. Since Java was designed for classes, a direct classless programming style is verbose and unnatural. However, annotation-driven code generation is enough to overcome this difficulty and the resulting programming style encourages modularity, composability and reusability. In that sense, we promote object interfaces as being both units of reuse and object factories. Our practical experience shows that separating the two notions leads to lots of boilerplate code, and is quite limiting when multiple inheritance with state is required. Abstract state operations avoid the key difficulties associated with multiple inheritance and state, while still being quite expressive. Moreover the ability to support constructors adds expressivity, which is not available in approaches such as Scala's traits/mixins.

Automatic generation of getters and setters is an old idea used in languages such as Self [21], Dart [1] and Newspeak [6]. The programmers specifies field signatures and (critically) the intention of storing such information, then the language generates getters and setters. Once state is abstracted away, it is well known that state access can be replaced with computation, but the type of the field stays the same. We do the opposite, the idea of the field is generated starting from signatures of getters and setters. In our approach, the intention of storing the information is not expressed by the programmer and set in stone but can vary by inheritance. In this case, the underlying type of the field can be changed by our fluid state, and `with` methods provide the right injection from the old type to the new.

ThisType and MyType Object interfaces support automatic type-refinement. Type refinement is part of a bigger topic in class-based languages: expressing and preserving type recursion and (nominal/structural) subtyping at the same time. One famous attempt in this direction is *MyType* [7], representing the type of **this**, changing its meaning along with inheritance. However when invoking a method with MyType in parameter positions, the exact type of the receiver must be known. This is a big limitation in class-based OO programming and is exasperated by the interface-based programming we propose: no type is ever going to be exact since classes are not explicitly used. A recent article [18] proposes two new features: exact statements and nonheritable methods. Both are related to our work: 1) any method generated inside the `of` method is indeed non-inheritable since there is no class name to extend from; 2) exact statements (a form of wild-card capture on the exact run-time type) could capture the "exact type" of an object even in a class-less environment. Admittedly, MyType greatly enhances the expressivity and extensibility of object-oriented programming languages. Object interfaces simulate some uses of MyType. But this approach only works for refining return types, whereas MyType is more general, as it also works for parameter types. Our approach to covariantly refine state can recover some of the additional expressivity of MyType. As illustrated with our examples, object interfaces are still very useful in many practical applications, yet they do not require additional complexity from the type system.

6. Conclusion

From Java 8, static and default methods are allowed in interfaces, which enable implementations inside interfaces. An important positive consequence that was probably overlooked is that the concept of class (in Java) is now (almost) redundant and unneeded. We propose a programming style, called Classless Java, where truly object-oriented programs and (reusable) libraries can be defined and used without ever defining a single class.

However, using this programming style directly in Java is very verbose. To avoid syntactic boilerplate caused by Java not being originally designed to work without classes, we introduce the `@Obj` annotation that provides default implementations for various methods (e.g. getters, setters, with-methods) and a mechanism to instantiate objects. We leverage on annotation processing and the Lombok library, in this way `@Obj` is just a normal Java library. The `@Obj` annotation helps programmers to write less cumbersome code while coding in Classless Java. Indeed, we think the obtained gain is so high that Classless Java with the `@Obj` annotation can be less cumbersome than full Java.

Acknowledgments

We would like to thank the reviewers for their helpful comments. This work is sponsored by the Hong Kong Research Grant Council Early Career Scheme project number 27200514.

References

[1] Dart programming language. `https://www.dartlang.org`, 2016.

[2] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(02):91–132, 2002.

[3] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. Traitrecordj: A programming language with traits and records. *Sci. Comput. Program.*, 78(5):521–541, 2013.

[4] V. Bono, E. Mensa, and M. Naddeo. Trait-oriented programming in java 8. In *PPPJ'14*, 2014.

[5] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.

[6] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, and E. Miranda. The newspeak programming platform, 2008.

[7] K. B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(02):127–206, 1994.

[8] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, 1998.

[9] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.

[10] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL'06*, 2006.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[12] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[13] L. Liquori and A. Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30 (2):11, 2008.

[14] D. Malayeri and J. Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA '09*, 2009.

[15] Neildo. Project lombok: Creating custom transformations. `http://notatube.blogspot.hk/2010/12/project-lombok-creating-custom.html`, 2011.

[16] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[17] J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *International workshop on foundations and developments of object-oriented languages*, 2006.

[18] C. Saito and A. Igarashi. Matching mytype to subtyping. *Sci. Comput. Program.*, 78(7):933–952, 2013.

[19] M. Sakkinen. Disciplined inheritance. In *ECOOP'89*, 1989.

[20] N. Scharli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP'03*, 2003.

[21] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA '87*, 1987.

[22] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, 1987.

[23] P. Wadler. The Expression Problem. Email, Nov. 1998. Discussion on the Java Genericity mailing list.

[24] Y. Wang and B. C. d. S. Oliveira. The expression problem, trivially! In *Proceedings of the 15th International Conference on Modularity*, 2016.

[25] Y. Wang, H. Zhang, B. C. d. S. Oliveira, and M. Servetto. Classless java - interface-based programming for the masses. Technical report, The University of Hong Kong, 2016.

[26] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[27] M. Zenger and M. Odersky. Independently extensible solutions to the expression problem. In *FOOL'05*, 2005.

[28] E. Zucca, M. Servetto, and G. Lagorio. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP'09*, 2009.

[29] R. Zwitserloot and R. Spilker. Project lombok. `http://projectlombok.org`, 2016.