Classless Java

Interface-Based Programming for the Masses

Abstract

This paper introduces interface-based object-oriented programming (in short IB). Unlike class based OO programming, in IB there is no need for classes. To support state, IB does not use fields directly. Instead it uses *abstract state operations*, such as getters, setters or clone-like methods. This new way to deal with state allows flexibility not typically available in CB. In IB state (including mutable state) can be type-refined in subtypes. The combination of a purely IB style of programming and type-refinement enables powerful idioms using multiple inheritance and state. To introduce IB to programmers we created Classless Java: an embeding of IB directly into Java. Classless Java uses annotation processing for code generation, and relies on new features of Java 8 for interfaces. We illustrate the usefulness of IB through various examples and case studies.

1. Introduction

Object-oriented languages strive to offer great code reuse. They aim to couple flexibility and rigour, expressive power and modular reasoning. Two main ideas emerged to this end: prototype-based (PB) [21] and class-based (CB) languages such as Java, C# or Scala. In prototype-based languages objects inherit from other objects. Thus objects own both behaviour 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. Here objects own the state, while classes contain behaviour and the structure of the state.

This paper presents a third alternative: the concept of **interface-based** object-oriented programming languages (in short IB), where objects implement interfaces directly. In IB interfaces own the implementation for the behaviour, 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 CB. Instantiation of objects is done using static factory methods in object interfaces.

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 the only responsible to define the ultimate behaviour of a method and, for example, if such a method is just a setter. 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 be only grown by inheritance. In contrast, in IB the state is never fixed, and methods that look like abstract setters and getters can always receive an explicit implementation down in the inheritance chain, improving **modularity and flexibility**. That is, the concept of abstract state is more fluid.

In the presence of subtyping, abstract state operations often require special care, as their types need to be refined. Object interfaces provide support for type-refinement and can automatically produce code that deals with type-refinement adequately. In contrast, verbose explicit type-refinement is typically required in CB. We believe that such verbosity hindered and slowed down the discovery of useful programming patterns involving type-refinement. In particular, a recently discovered solution [23] to the Expression Problem [22] 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 much more obvious.

One important advantage of the use 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* "fields" in subtypes. This is typically forbidden in CB: it is widely known that *naive* type-refinement of mutable fields is not type-safe. While covariant refinement of mutable fields is supported by some type systems [5, 6, 9, 18], this requires significant complexity and restrictions to ensure that all uses of covariant state are indeed type-safe.

Finally, another advantage of IB is its support for multiple inheritance. The literature provides good examples on how easy and modular it is to combine multiple sources of pure behaviour, using mechanisms such as traits [20]. In Java multiple *interface* inheritance has been supported since inception, and in Java 8 default methods [11] bring some of the advantages of traits into Java. In contrast, the literature [4, 14, 19] is also rich on how hard it is to modularly combine multiple sources of behaviour **and** state with multiple *implementation*

inheritance of classes. In IB there is only multiple interface inheritance, yet programmers can still use state via the abstract state operations. IB enables powerful idioms using multiple inheritance and state.

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 for interfaces: 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[1]. We call this restricted version of Java *Classless Java*.

Using Java annotation processors, we produce an implementation of Classless Java, which allow us to stick to pure Java 8. By annotating with `@Obj` the interfaces that represent object interfaces, we can automatically generate code for interface instantiation and type refinement. Such code should not be needed in the first place in a real IB language, and the annotation processor allows us to transparently hide it from Java programmers. 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. As a result, we experiment object interfaces with several interesting Java programs, and conduct various case studies. Finally, we also formally define the behaviour of our `@Obj` annotation and discuss its properties.

While the Java embedding has obvious advantages from the practical point-of-view, it also imposes some limitations that a new IB language would not have. In particular, supporting proper encapsulation is difficult in Java due to limitations of Java interfaces. We discuss these limitations, possible workarounds, and native language support in Section 6.

In summary, the contributions of this paper are:

- **IB and Object Interfaces:** a novel take on object orientation, allowing 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. We provide examples, informal and formal descriptions of Classless Java. Existing Java projects can use our approach and still be backward compatible with their client, in a way that is formally 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] More precisely, we never use the **`class`** keyword.

2. A Running Example: Animals

This section illustrates how our programming style, supported by the `@Obj` annotation, enables powerful programming idioms based on multiple inheritance and type refinements. We propose a standard example: `Animal`s with a two-dimensional `Point2D` representing their `location`. Some kinds of animals are `Horse`s and `Bird`s. Birds can `fly`, thus their locations need to be three-dimensional `Point3D`s (field type refinement). Finally, we model `Pegasus` (one of the best-known creatures in Greek mythology) as a kind of `Animal` with the skills of both `Horse`s and `Bird`s (multiple inheritance). A simple class diagram illustrating the basic system is given on the left side of Figure 1.[2]

2.1 Simple Multiple Inheritance with Default Methods

Before modelling the complete animal system, we start with a simplified 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 in Java 8 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 behaviour 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:

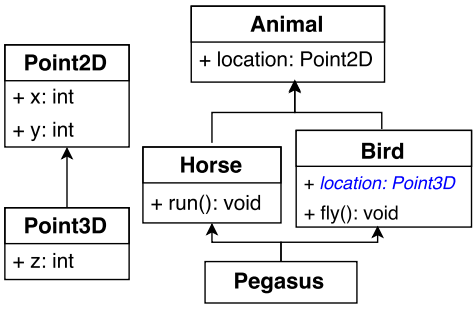
```
class HorseImpl implements Horse {}
class BirdImpl implements Bird {}
class PegasusImpl implements Pegasus {}
```

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 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

[2] Some research argues in favor of using subtyping for modeling taxonomies, other research argues against this practice, we do not wish to take sides in this argument, but to provide an engaging example.




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

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

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:

```
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:

```
@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:

```
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:

```
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 a user, who only sees 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 using coordinates.

`Point2D`: simple immutable data with fields Here we will illustrate how points are modelled with interfaces. In IB we do not talk about state directly. Instead 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:

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

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

```
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 in this case:

```
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 can be automatically 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. That is, we can just write

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

More precisely, a field or factory parameter is generated for every abstract method that takes no parameters (except for methods with special names). An example of using `Point2D` is:

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

where we return a new point, using `42` as x-coordinate, and taking all the other information (only `y` in this case) from another point.

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. For example:

```
@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:

```
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*/ }
   default Point2D withY(int val){ /*myCode*/ } }
```

is expanded into

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

Only code for methods needing implementation is generated. Thus, programmers can easily customize the behaviour 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 immutable data structure to model them. However 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 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 (all generated automatically) provide a basic interface for dealing with animals.

In summary, dealing with 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, with object interfaces, 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 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 Just to give a feeling of how much mechanical code `@Obj` is generating, we show the generated code for the `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`**.

```
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.

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. Furthermore object interfaces support direct object intantiation via `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 5.2.

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 or a 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 to achieve a similar level of integration and tool support as Java. To have an approach that can both illustrate IB programming and be practical, we have instead implemented `@Obj` as an annotation over Java 8, and a *compi-*

	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)`	Update all matching fields in val.

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

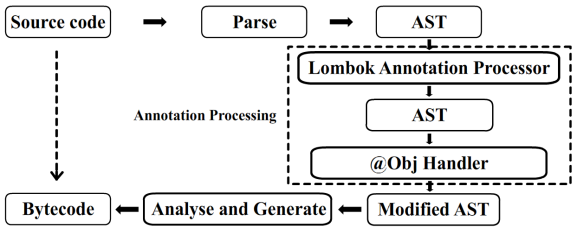


Figure 4. The flow chart of `@Obj` annotation processing.

lation agent. That is, the Classless Java style of programming is supported by a Java 8 library.

Disciplined use of Classless Java (that is 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's support. The key to our implementation is the use of compilation agents, which allows us to rewrite the Java AST just before compilation. We discuss 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 the abstract syntax tree and the Java bytecode.

This process is facilitated by frameworks like Lombok [27]: a Java library that aims at removing (or 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.

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 of course, 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 do not support. For example, the standard `javax.annotation` processor, which is part of the Java platform, only allows generation of *new code*, and the new code has to be written in *new files*. Modification and/or reinterpretation of existing code is not supported.

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 the need of being reprocessed and recompiled, making our approach 100% compatible with existing Java libraries, which can be used and extended normally. Of course, Java libraries can also receive and use instances of object interfaces as normal objects.

Tool support Features written in Lombok integrate and are supported directly in the language, and are often also supported by (most) tools. For example 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 as well as the generated ones. In Figure 5, `@Obj` generates an `of` method in `Point2D`, and `of`, `withX`, `withY` methods in `Point3D`. These methods are also visible to users, showing their types and modifiers in Outline window. Moreover, as a useful IDE feature, the auto-completion also works for these newly generated methods.

Clarity against obfuscation Preprocessors bring great power, which can easily be misused producing code particularly hard to understand. Thus code quality and maintainability are reduced. Compilation agents start from Java syntax, but they can reinterpret it. Preserving the syntax avoids syntactic conflicts, and allows many tools to work transparently.

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 behaviour of abstract methods; add method implementations; and refine return types. We consider this to

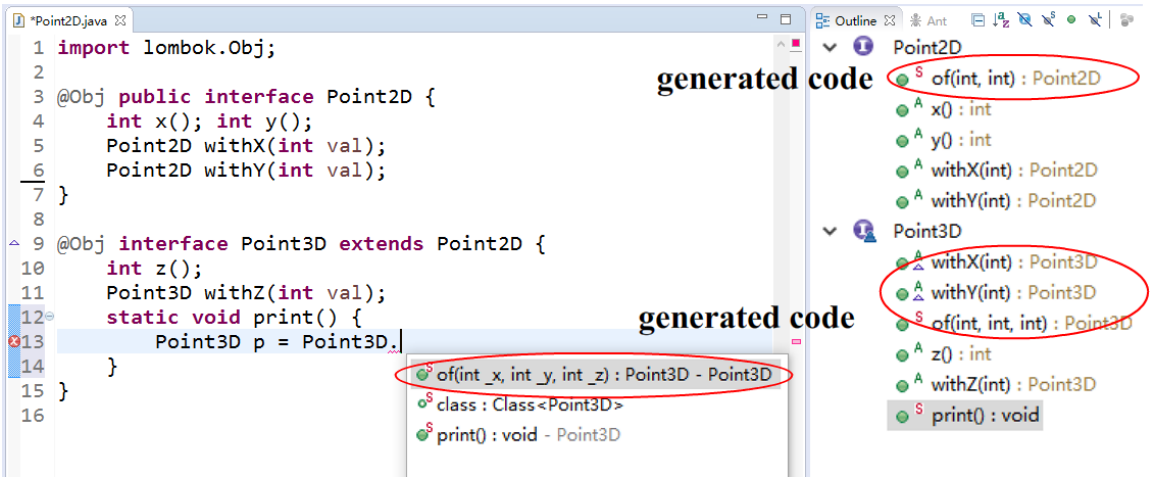


Figure 5. Generated methods shown in the Outline window of Eclipse and auto-completion.

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 some 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. That is, type errors are not present in code the user has not written (for example `of` methods in Figure 5). 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 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 behaviour 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 would need to give up automatic type refinement, which is an essential part of IB programming. However, the reader should note that Java libraries almost always break heir coherence during evolution and still claim backward compatibility (false in theory but statistically true in practice). 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.

Section 4 provide a formal definition for our safety claims.

3.3 Limitations

Our prototype implementation has certain limitations:

- Eclipse has its own compiler (ejc), alternative to javac. Lombok allows writing handlers for either javac or ejc. At this stage our implementation only realizes the Eclipse handler and our experiments are all conducted in Eclipse. The implementation for the `javac` handler is still missing.
- The current implementation supports generics in a simplified manner: type parameters can be used but the generic method typing is neither formalized in this paper nor explicitly checked by our implementation, but simply delegated to the Java compiler.
- Lombok offers only limited/experimental support for separate compilation, that is to access information of code defined in different files. In the same spirit, we have a mature `@Obj` annotation, which does not support separate compilation yet: it requires that all related interfaces have to appear in a single Java file. Reusing the logic inside the yet experimental Lombok annotation `@Delegate`, we

also offer a less polished annotation supporting separate compilation for files in the same package.

4. What `@Obj` Generates

This section gives an overview of what `@Obj` generates, and what formal properties are guaranteed in the translation.

We formalize syntax and typing for Classless Java in Appendix A, which models the essence of Java interfaces with default methods. Classless Java is just a proper subset of Java 8, so it is easy to understand the translation presented in this section without the syntax and typing rules of Classless Java. Since the formalized part of Classless Java does not consider casts or `instanceof`, the `with` method is not included in the formal translation. For the same reason **`void`** returning setters are not included, since they are just a minor variation over the more interesting fluent setters, and they would require special handling just for the conventional **`void`** type. Since our properties are about preserving typing, we do not need to formalize Classless Java semantics to prove our statements.

4.1 Translation

For the purposes of the formalization, the translation is divided into two parts for more convenient discussion on formal properties later. To this aim we introduce the annotation `@ObjOf`. Its role is only in the translation process, hence is not part of the Classless Java language. `@ObjOf` generates the constructor method `of`, while `@Obj` automatically refines the return types and calls `@ObjOf`.

Figure 6 presents the translation. In the first function, `@Obj` injects refined methods to interface I_0. The second function, `@ObjOf` invokes $\mathsf{ofMethod}(I_0)$ and generates the `of` method for I_0, if such a method does not exist in its domain, and all the abstract methods are valid for the annotation.

Figure 7 presents more details on the auxiliary functions. The first two points of Figure 7 define function refine. This function generates unimplemented `with-` and fluent setters in the interface, where the return types have been refined. To determine whether a method needs to be generated, we check if such `with-` or setter methods require an implementation in I_0, but are not declared directly in I_0. The third point gives the definition of valid: it is valid to annotate an interface if all abstract methods (that is, all those requiring an implementation) are valid. That is, we can categorize them in a pattern that we know how to implement (right column): it is either a field getter (first point), a with method (second point) or a setter (third point). Note that we write `with#`*m* to append *m* to `with`, following the camelCase rule. The first letter of *m* must be lower-case and is changed to upper-case upon appending. For example `with#foo=withFoo`. Special names $\mathsf{special}(m)$ are `with` and all identifiers of the form `with#`*m*.

Figure 8 defines the ofMethod function, which generates the static method `of` as an object factory. It detects all the field methods of I_0 and use them to synthesize its arguments. The return statement instantiates an anonymous class which generates the needed getters, fluent setters and with-methods. The right column first point collects the getter methods, the second and third point generate implementations for `with-` methods if needed; similarly, the fourth and fifth point generate fluent setters if needed.

Some other features of `@Obj`, including non-fluent setters and the `with` method are not formalized here. Appendix B.2 gives an detailed but informal explanation of generation for those methods.

4.2 Results

Classless Java provides some guarantees regarding the generated code. Essentially, Classless Java ensures the *self coherence* and *client coherence* properties informally introduced in Section 3. Furthermore, we can show that *if there are no type-refinements*, then *heir coherence* also holds. The result about heir coherence is possible to prove because the translation is split into two parts. In essence heir coherence is a property of the translation of `@ObjOf`, but not of `@Obj`.

To formally characterize the behavior of our annotation and the two levels of guarantees that we offer, we provide some notations and two theorems:

- We denote with $I^{\mathcal{I}}$ and m^{meth} the name of an interface and of a method.
- An interface table IT is OK if under such interface table, all interfaces are OK, that is, well typed.
- Since interface tables are just represented as sequences of interfaces we write IT = $\mathcal{I}$ IT' to select a specific interface in a table.
- IT contains an heir of *I* if there is an interface that extends it, or a **`new`** that instantiates it.

Theorem 1 (@ObjOf)**.** *If a given interface table $\mathcal{I}$ IT is OK where $\mathcal{I}$ has `@ObjOf`, $\mathsf{valid}(I^{\mathcal{I}})$ and `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, then the interface table $[\![\mathcal{I}]\!]$ IT is OK.*

Theorem 2 (@Obj)**.** *If a given interface table $\mathcal{I}$ IT is OK where $\mathcal{I}$ has `@Obj`, $\mathsf{valid}(I^{\mathcal{I}})$ and `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, and there is no heir of $I^{\mathcal{I}}$, then the interface table $[\![\mathcal{I}]\!]$ IT is OK.*

Informally, the theorems mean that for a client program that type-checks before the translation is applied, if the annotated type has no subtypes and no objects of that type are created, then type safety of the generated code is guaranteed after the successful translation.

The second step of `@Obj`, namely what `@ObjOf` does in the formalization, is guaranteed to be type-safe for the three kinds of coherence by the `@ObjOf` theorem. The `@Obj` theorem is more interesting: since `@Obj` does not guarantee heir coherence, we explicitly exclude the presence of heirs. In this way the `@Obj` theorem guarantees only self and client coherence. The formal theorem proofs are available in Appendix C.

Type preservation Note that we preferred to introduce self, client and heir coherence instead of referring to conventional type preservation theorems. The reason is to better model how our approach behaves in a object-oriented software ecosystem

- $[\![$`@Obj interface` I_0 `extends` $\overline{I}${ $\overline{meth}$ }$]\!]$ = $[\![$`@ObjOf interface` I_0 `extends` $\overline{I}${ $\overline{meth}$ $\overline{meth}'$ }$]\!]$
  with $\overline{meth}' = \mathsf{refine}(I_0, \overline{meth})$
- $[\![$`@ObjOf interface` I_0 `extends` $\overline{I}${ $\overline{meth}$ }$]\!]$ = `interface` I_0 `extends` $\overline{I}${ $\overline{meth}$ $\mathsf{ofMethod}(I_0)$}
  with $\mathsf{valid}(I_0), \texttt{of} \notin \mathsf{dom}(I_0)$

Figure 6. The translation functions of `@Obj` and `@ObjOf`.

- I_0 `with#`m`(`I `_val);` $\in \mathsf{refine}(I_0, \overline{meth})$ = $\mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m, I_0), I_0)$, `with#`$m \notin \mathsf{dom}(\overline{meth})$
- I_0 `_`m`(`I `_val);` $\in \mathsf{refine}(I_0, \overline{meth})$ = $\mathsf{isSetter}(\mathsf{mbody}(\_m, I_0), I_0)$, `_`$m \notin \mathsf{dom}(\overline{meth})$
- $\mathsf{valid}(I_0)$ = $\forall m \in \mathsf{dom}(I_0)$, if $mh;$ = $\mathsf{mbody}(m, I_0)$, one of the following cases is satisfied: $\mathsf{isField}(meth), \mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$

- $\mathsf{isField}(I\,m\texttt{();})$ = not $\mathsf{special}(m)$
- $\mathsf{isWith}(I'$ `with#`m`(`$I\,x$`);`$, I_0)$ = $I_0 <: I'$, $\mathsf{mbody}(m, I_0) = I\,m\texttt{();}$ and not $\mathsf{special}(m)$
- $\mathsf{isSetter}(I'$ `_`m`(`$I\,x$`);`$, I_0)$ = $I_0 <: I'$, $\mathsf{mbody}(m, I_0) = I\,m\texttt{();}$ and not $\mathsf{special}(m)$

Figure 7. The refine and valid functions (left) and auxiliary functions (right).

- $\mathsf{ofMethod}(I_0)$ = `static` I_0 `of(`I_1 `_`m_1`,` $\ldots$ I_n `_`m_n`){`
  `return new` I_0`(){`
  $I_1\ m_1 =$ `_`m_1`;` $\ldots$ $I_n\ m_n =$ `_`m_n`;`
  $I_1\ m_1$`(){return` m_1`;}` $\ldots$ $I_n\ m_n$`(){return` m_n`;}`
  $\mathsf{withMethod}(I_1, m_1, I_0, \overline{e}_1) \ldots \mathsf{withMethod}(I_n, m_n, I_0, \overline{e}_n)$
  $\mathsf{setterMethod}(I_1, m_1, I_0) \ldots \mathsf{setterMethod}(I_n, m_n, I_0)$
  `};}`
  with $I_1\ m_1\texttt{();}, \ldots I_n\ m_n\texttt{();} = \mathsf{fields}(I_0)$
  and $\overline{e}_i = m_1\texttt{,} \ldots \texttt{,}m_{i-1}\texttt{,\_val,}m_{i+1}\texttt{,} \ldots \texttt{,}m_n$

- $meth \in \mathsf{fields}(I_0)$ = $\mathsf{isField}(meth)$ and $meth = \mathsf{mbody}(m^{meth}, I_0)$
- $\mathsf{withMethod}(I, m, I_0, \overline{e})$ = I_0 `with#`m`(`I `_val){return` I_0`.of(`$\overline{e}$`);}`
  with $\mathsf{mbody}(\texttt{with\#}m, I_0)$ having the form $mh;$
- $\mathsf{withMethod}(I, m, I_0, \overline{e})$ = $\emptyset$ otherwise
- $\mathsf{setterMethod}(I, m, I_0)$ = I_0 `_`m`(`I `_val){`m`= _val;return this;}`
  with $\mathsf{mbody}(\_m, I_0)$ having the form $mh;$
- $\mathsf{setterMethod}(I, m, I_0)$ = $\emptyset$ otherwise

Figure 8. The generated `of` method (left) and auxiliary functions (right).

with inheritance, where only some units may be translated/-expanded. Note inheritance's crucial influence in heir coherence. Our formulation of client coherence allows us to discuss about intermediate stages where only some code units are translated/expanded. Conventional type preservation refers only to completely translated program. Our coherence guarantees mean that developers and designers of Java libraries and frameworks can start using IB (and our `@Obj` annotation) in the evolution of their products and still retain backward compatibility with their clients.

5. 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 conviniently encoded in Classless Java. The second application shows how to model embedded DSLs based on fluent APIs. The two larger case studies, take existing projects and refactor them into Classless Java. The first case study shows a significant reduction in code size, while the second case study maintains the same amount of code, but improves modularity.

5.1 The Expression Problem with Object Interfaces

As a 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) [22], which is a well-known problem about modular extensibility issues in software evolution. Recently, a new solution [23] using only covariant type refinement was proposed. When this solution is modeled with interfaces and default methods, it can even provide independent extensibility [25]: 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 5.4, 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 and addition, and an initial operation `eval` for expression evaluation. As shown in Figure 9, `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();
```

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

```
 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 9 (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 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, we first define a new operation `collectLit`, which collects all literal components in an expression. For space reasons, we omit the definitions of the methods:

```
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 (`print` and `collectLit`) together:

```
interface ExpPC extends ExpP, ExpC {}
```

`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`.

Note that the code for instantiation is automatically generated by `@Obj`. Creating a simple expression of type `ExpPC` is as simple as:

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

Without our approach, tedious instantiation code would need to be defined manually.

5.2 Embedded DSLs with Fluent Interfaces

Since the style of fluent interfaces was invented in Smalltalk as method cascading, more and more languages came to support fluent interfaces, including JavaScript, Java, C++, D, Ruby, Scala, etc. 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. Fluflu enables the creation of a fluent API and implements control over method chaining by using Java annotations. However methods that returns **this** are still hand-written.

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, and not the old `Database` type, which would prevent calling `orderBy`.

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

5.3 A Maze Game

This case study is a simplified variant of a Maze game, which is often used [3, 10] 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 [3], and the code for that implementation is available online. We reimplemented the game using `@Obj`. Due to space constraints, we omit the code here. The following table summarizes the number of lines of code and classes/interfaces in each implementation:

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

The `@Obj` annotation allowed us to reduce the interfaces/-classes used in Bono et al.'s implementation by 21.4% (from 14 to 11). The reductions were 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. To ensure a fair comparison, we used the same coding style as Bono et al.'s.

5.4 Refactoring an Interpreter

The last case study refactors the code from an interpreter for a Lisp-like language `Mumbler`[3], which is created as a tutorial for the Truffle Framework [24]. 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, would normally require changing the existing code base directly.

Our refactoring applies the pattern presented in Section 5.1 to the existing Mumbler code base to improve its modularity and extensibility. Using the refactored code base it is 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. In the refactored code base the pretty printer is added modularly. Although the code in the refactored version is slightly increased (by 2.2% SLOC), the modularity is greatly increased, allowing for improved reusability and maintainability.

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

6. Future work

Encapsulation A limitation of our approach, inherited from Java, is the lack of some support for encapsulation: in Java interfaces all methods are public. However, it is possible to workaround the limitations of interfaces by a variation of the facade pattern (as suggested by [2]):

```
@Obj interface InternalData implements ExposedData {
   /*all the methods*/} //package visibility
public interface ExposedData {
 /*just the exposed interface*/
 static ExposedData of(...) {
   return InternalData.of(...);} }
```

This is a safe and explicit way to list all the public interfaces and to hide all the implementation details from outside of the package.[4] Using packages as a boundary we can make all object interfaces *where some form of privateness is desirable* as invisible types for the rest of the program. This preserves encapsulation, but limits inheritance: package protected interfaces cannot be seen, thus cannot be implemented from outside the package. This can raise questions: keeping fields private is a staple of CB. By forcing the user/heir to rely only on methods, we avoid (a part) of the fragile base class problem and we obtain representation independence. In IB fields do not exists; abstract state methods like getters and setters are abstract: it does not make sense to make them private; however, since they are methods, the base interface may be able to evolve freely. To understand if a fragile base interface problem exists in IB, we need more research.

A real IB language In this paper we presented the concept of IB as a programming pattern over Java. An alternative would be to have a language providing IB as its only paradigm. We do not wish to define here a complete IB language, just to show the reader IB as a promising way to design a new OO language. To define such language, the first task is to define the set of abstract state operations. Of course we need getters; `withX` methods are also convenient, and they enable a clean functional programming style that would be otherwise too cumbersome. Currently we are offering both setters and fluent setters. We believe fluent setters to be sufficient; we could drop the concept of (non fluent)setters. Manually defining abstract state methods can be verbose, we could offer a convenient syntax to declare in a compact way get/set/with. For example:

```
obj interface Foo{
```

[4] Note how `ExposedData.of` does not need to map directly to the fields, and can also perform invariant checking.

```
  String bar;// expanded as getter and with:
  //String bar(); Foo withBar(String val);
  var String beer;//as before plus fluent setter
}
```

It should be possible to offer pre initialized abstract state, for example

```
obj interface Foo{ var String bar="hello"; }
```

could be a short hand for writing

```
obj interface Foo{
  String bar();
  Foo withBar(String val);
  Foo bar(String val);
  String initBar(){return "hello";}
}
```

and the automatically generated `of` method will use `initX` methods to initialize fields instead of generating the corresponding parameter.

We may also want to offer a compact syntax for property updaters and generalized with methods;

```
obj interface Foo{
  updater Bar, Beer;//declares imperative updaters:
  //Foo set(Bar val); Foo set(Beer val);
  with Foo, Beer;//declared functional updaters:
  //Foo with(Bar val); Foo with(Beer val);
}
```

Finally, certain methods may have a specific meaning. For example if a method called `postInit` is present, it will be called after construction. It may be handled specially during multiple implementations, so that all relevant `postInit` methods are called in some order.

7. Related Work

Models of object-oriented programming There are traditionally two big styles of OOP: class-based and prototype-based. The class-based tradition started with Simula [7], and is the dominant style of OOP today. Major languages such as Java, C#, C++ or Scala favour a class-based OOP style. Very often class-based languages are statically typed and they have interfaces or some other similar mechanisms such as traits. However, in contrast to IB, interfaces or traits in CB are not meant to replace classes, but rather, they complement classes. Normally interfaces or traits define the signatures of the methods and *may* have some behaviour (such as in Java 8 interfaces or Scala traits). However classes are still responsible for object instantiation and handling state. Prototype-based languages started with Self [21], and are still widely used today via languages like JavaScript. Frequently prototype-based languages are dynamically typed. IB is clearly closer to CB. However the key difference is that classes are not needed in IB. Object interfaces recover the responsibility of object instantiation, and abstract state operations provide a way to deal with state. A primary motivation of IB is to improve modularity and flexibility. Central to these goals are the more expressive and automated forms of type-refinement, and support for multiple interface inheritance.

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, and can cause difficulties (including the famous diamond (fork-join) problem [4, 19], conflicting methods, etc.) in reasoning about programs. To conciliate the need for expressive power and simplicity, many models have been proposed, including C++ virtual inheritance, mixins [4], 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 [8]. However C++ suffers from the 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. Moreover, customized factory methods are also allowed: if users are not satisfied with the default generated `of` method, they can implement their own.

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 this limitation, an algebra of mixin operators is introduced [1], but this raises the complexity, especially when constructors and fields are considered [26]. 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 [3] 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 to be an alternative to traits or mixins. In the trait model two concepts (traits and classes) coexist and cooperate. Some authors [2] see this as good language design fostering good software development by helping programmers to think about the structure of their programs. However, other authors see the need of two concepts and the absence of state as drawbacks of this model [14]. Object interfaces are units of reuse, and at the same time they provide factory methods for instantiation and support state. Our approach promotes the use of interfaces instead of classes, in order to rely on 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, by keeping a strong OO feel. In that sense, we promote object interfaces as being both units of reusable code and object factories. Our practical experience is that, in Java, separating the two notions leads to a lot 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.

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 provided by *MyType* [5], representing the type of **this**, changing its meaning along with inheritance. However when invoking a method with MyType in a parameter position, the exact type of the receiver must be known. This is a big limitation in class-based object-oriented 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] lights up this topic, proposing 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 wildcard 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 expressiveness and extensibility of object-oriented programming languages. Object interfaces use covariant-return types to 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 expressiveness of MyType. As illustrated with our examples and case studies, object interfaces are still very useful in many practical applications, yet they do not require additional complexity from the type system.

8. Conclusion

Before Java 8, concrete methods and static methods were not allowed to appear in interfaces. Java 8 allows static interface methods and introduces *default methods*, which enables implementations inside interfaces. This had an important positive consequence that was probably overlooked: the concept of class (in Java) is now (almost) redundant and unneeded. We proposed 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 a new annotation, `@Obj`, which 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; thus our proposed style can be integrated in any Java project. 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.

Classless Java is just a programming style, but is showing the way for a new flavour of object orientation: We propose interface-based object-oriented languages (IB), as opposed to class-based or prototype-based. In IB state is not modelled at the platonic/ideal level but is handled excursively by instances. This unlocks useful code reuse patterns, as shown in Section 2. An interesting avenue for future work would be to design a new language based on IB. With a proper language design we would not need to restrict ourselfves to the limitations of Java and its syntax.

References

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

[2] 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.

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

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

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

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

[7] O.-J. Dahl and K. Nygaard. Simula: An algol-based simulation language. *Commun. ACM*, 9(9), Sept. 1966.

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

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

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

[11] B. Goetz and R. Field. Featherweight defenders: A formal model for virtual extension methods in java, 2012. http://cr.openjdk.java.net/ briangoetz/lambda/featherweight-defenders.pdf.

[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, 2011. http://notatube.blogspot.hk/2010/12/project-lombok-creating-custom.html.

[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 *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, 1987.

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

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

[24] 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.

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

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

[27] R. Zwitserloot and R. Spilker. Project lombok. http://projectlombok.org.

A. Formal Semantics

This section presents a formalization of Classless Java, which models the essence of Java interfaces with default methods. This formalization is used to define the semantics of object interfaces.

A.1 Syntax

Figure 10 shows the syntax of Classless Java. The syntax formalizes a minimal subset of Java 8, focusing on interfaces, default methods and object creation literals. There is no syntax for classes. To help readability we use many metavariables to represent identifiers: C, x, f and m; however they all map to a single set of identifiers as in Java. Expressions consist of conventional constructs such as variables (x), method calls ($e.m(\overline{e})$) and static method calls ($I.m(\overline{e})$). For simplicity the degenerate case of calling a static method over the **this** receiver is not considered. A more interesting type of expressions is super calls ($I.$**super**$.m(\overline{e})$), whose semantics is to call the (non-static) method m over the **this** receiver, but statically dispatching to the version of the method as visible in the interface I. A simple form of field updates (x=e;e') is also modeled. In the syntax of field updates x is expected to be a field name. After updating the field x using the value of e, the expression e' is executed. To blend the statement based nature of Java and the expression based nature of our language, we consider a method body of the form **return** x=e;e' to represent x=e;**return** e' in Java. Finally, there is an object initialization expression from an interface I, where (for simplicity) all the fields are initialized with a variable present in scope. To be fully compatible with Java, the concrete syntax for an interface declaration with empty supertype list would also omit the **extends** keyword. Following standard practice, we consider a global Interface Table (IT) mapping from interface names I to interface declarations $\mathcal{I}$.

The environment Γ is a mapping from variables to types. As usual, we allow a functional notation for Γ to do variable lookup. Moreover, to help us define auxiliary functions, a functional notation is also allowed for a set of methods $\overline{meth}$, using the method name m as a key. That is, we define $\overline{meth}(m) = meth$ iff there is a unique $meth \in \overline{meth}$ whose name is m. For convenience, we define $\overline{meth}(m) = \mathsf{None}$ otherwise; moreover $m \in \mathsf{dom}(\overline{meth})$ iff $\overline{meth}(m) = meth$. For simplicity, we do not model overloading, thus for an interface to be well formed its methods must be uniquely identified by their names.

A.2 Typing

Typing statement $\Gamma \vdash e \in I$ reads "in the environment Γ, expression e has type I.". Before discussing the typing rules we discuss some of the used notation. As a shortcut, we write $\Gamma \vdash e \in I <: I'$ instead of $\Gamma \vdash e \in I$ and $I <: I'$.

We omit the definition of the usual traditional subtyping relation between interfaces, that is the transitive and reflexive closure of the declared **extends** relation. The auxiliary notation Γ^{mh} trivially extracts the environment from a method header, by collecting the all types and names of the method parameters. The notation m^{mh} and I^{mh} denotes respectively, extracting the method name and the return type from a method header. $\mathsf{mbody}(m, I)$, defined in Appendix A.3, returns the full method declaration as seen by I, that is the method m can be declared in I or inherited from another interface. $\mathsf{mtype}(m, I)$ and $\mathsf{mtypeS}(m, I)$ return the type signature from a method (using $\mathsf{mbody}(m, I)$ internally). $\mathsf{mtype}(m, I)$ is defined only for non static methods, while $\mathsf{mtypeS}(m, I)$ only for static ones. We use $\mathsf{dom}(I)$ to denote the set of methods that are defined for type I, that is: $m \in \mathsf{dom}(I)$ iff $\mathsf{mbody}(m, I) = meth$.

In Figure 11 we show the typing rules. We discuss the most interesting rules, that is (T-OBJ) and (T-INTF). Rule (T-OBJ) is the most complex typing rule. Firstly, we need to ensure that all field initializations are type correct, by looking up the type of each variable assigned to a field in the typing environment and verifying that such type is a subtype of the field type. Finally, we check that all method bodies are well-typed. To do this the environment used to check the method body needs to be extended appropriately: we add all fields and their types; add **this** : I; and add the arguments (and types) of the respective method. Now we need to check if the object is a valid extension for that specific interface. This can be logically divided into two steps. First we check that all method headers are valid with respect to the corresponding method already present in I:

- $\mathsf{sigvalid}(mh_1 \dots mh_n, I) =$
  $\forall i \in 1..n \;\; mh_i; <: \mathsf{mbody}(m^{mh_i}, I)$

Here we require that for all newly declared methods, there is a method with the same name defined in the interface I, and that such method is a supertype of the newly introduced one. We define subtyping between methods in a general form that will also be useful later.

- $I\, m(I_1 x_1 \dots I_n x_n)\,; <: I' m(I_1 x'_1 \dots I_n x'_n)\,; \quad = \quad I <: I'$
- $meth <:$ **default** mh**{return** _**;}** $\quad = \quad meth <: mh;$
- **default** mh**{return** _**;}** $<: meth \quad = \quad mh; <: meth$

We allow return type specialization as introduced in Java 5. A method header with return type I is a subtype of another method header with return type I' if all parameter types are the same, and $I <: I'$. A default method $meth_1$ is a subtype of another default method $meth_2$ iff mh^{meth_1} is a subtype of mh^{meth_2}. Secondly, we check that all abstract methods (which need to be explicitly overridden) in the interface have been implemented:

- $\mathsf{alldefined}(mh_1 \dots mh_n, I) \quad = \quad \forall m$ such that
  $\mathsf{mbody}(m, I) = mh; \exists i \in 1..n \; m^{mh_i} = m$

The rule (T-INTF) checks that an interface I is correctly typed. First we check that the body of all default and static methods are well typed. Then we check that $\mathsf{dom}(I)$ is the same as $\mathsf{dom}(I_1) \cup \dots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$. This is not a trivial check, since $\mathsf{dom}(I)$ is defined using mbody, which would be undefined in many cases: notably if a method $meth \in \overline{meth}$ is not compatible with some method in

e	::=	$x \mid e.m(\overline{e}) \mid I.m(\overline{e}) \mid I.\texttt{super}.m(\overline{e}) \mid x{=}e;e' \mid obj$	expressions
obj	::=	$\texttt{new}\, I(\,)\{\, \overline{field}\; mh_1\{\, \texttt{return}\; e_1;\}\; \ldots\; mh_n\{\, \texttt{return}\; e_n;\}\}$	object creation
$field$	::=	$I\, f{=}x;$	field declaration
$\mathcal{I}$	::=	$ann\; \texttt{interface}\; I\; \texttt{extends}\; \overline{I}\{\, \overline{meth}\,\}$	interface declaration
$meth$	::=	$\texttt{static}\; mh\{\, \texttt{return}\; e;\} \mid \texttt{default}\; mh\{\, \texttt{return}\; e;\} \mid mh;$	method declaration
mh	::=	$I_0\; m\; (I_1\; x_1 \ldots I_n\; x_n)$	method header
ann	::=	$\texttt{@Obj} \mid \emptyset$	annotations
Γ	::=	$x_1{:}I_1 \ldots x_n{:}I_n$	environment

Figure 10. Grammar of Classless Java

$$\text{(T-INVK)}\quad \frac{\Gamma \vdash e \in I_0 \quad \forall i \in 1..n\;\; \Gamma \vdash e_i \in \_ <: I_i \quad \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash e.m(e_1 \ldots e_n) \in I}$$

$$\text{(T-STATICINVK)}\quad \frac{\forall i \in 1..n\;\; \Gamma \vdash e_i \in \_ <: I_i \quad \mathsf{mtypeS}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash I_0.m(e_1 \ldots e_n) \in I}$$

$$\text{(T-SUPERINVK)}\quad \frac{\Gamma(\texttt{this}) <: I_0 \quad \forall i \in 1..n\;\; \Gamma \vdash e_i \in \_ <: I_i \quad \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash I_0.\texttt{super}.m(e_1 \ldots e_n) \in I}$$

$$\text{(T-VAR)}\quad \frac{\Gamma(x) = I}{\Gamma \vdash x \in I}$$

$$\text{(T-OBJ)}\quad \frac{\begin{array}{c}\forall i \in 1..k\;\; \Gamma(x_i) <: I_i \\ \forall i \in 1..n\;\; \Gamma, f_1{:}I_1, \ldots, f_k{:}I_k, \texttt{this}{:}I, \Gamma^{mh_i} \vdash e_i \in \_ <: I^{mh_i} \\ \mathsf{sigvalid}(mh_1 \ldots mh_n, I) \qquad \mathsf{alldefined}(mh_1 \ldots mh_n, I)\end{array}}{\Gamma \vdash \texttt{new}\, I(\,)\{\, I_1\, f_1{=}x_1; \ldots I_k\, f_k{=}x_k;\; mh_1\{\, \texttt{return}\; e_1;\} \ldots mh_n\{\, \texttt{return}\; e_n;\}\} \in I}$$

$$\text{(T-UPDATE)}\quad \frac{\Gamma \vdash e \in \_ <: \Gamma(x) \quad \Gamma \vdash e' \in I}{\Gamma \vdash x{=}e;e' \in I}$$

$$\text{(T-INTF)}\quad \frac{\begin{array}{c}IT(I) = ann\; \texttt{interface}\; I\; \texttt{extends}\; I_1 \ldots I_n \{\, \overline{meth}\,\} \\ \forall \texttt{default}\; mh\{\, \texttt{return}\; e;\} \in \overline{meth},\;\; \Gamma^{mh}, \texttt{this}{:}I \vdash e \in \_ <: I^{mh} \\ \forall \texttt{static}\; mh\{\, \texttt{return}\; e;\} \in \overline{meth},\;\; \Gamma^{mh} \vdash e \in \_ <: I^{mh} \\ \mathsf{dom}(I) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})\end{array}}{I\ \text{OK}}$$

Figure 11. CJ Typing

$\mathsf{dom}(I_1) \ldots \mathsf{dom}(I_n)$ or if there are methods in any $\mathsf{dom}(I_i)$ and $\mathsf{dom}(I_j)$ $(i, j \in 1..n)$ conflict.

A.3 Auxiliary Definitions

Defining mbody is not trivial, and requires quite a lot of attention to the specific model of Java interfaces, and to how it differs w.r.t. Java Class model. $\mathsf{mbody}(m, I)$ denotes the actual method m (body included) that interface I owns. The method can either be defined originally in I or in its supertypes, and then passed to I via inheritance.

- $\mathsf{mbody}(m, I_0) = \mathsf{override}(\overline{meth}(m), \mathsf{needed}(m, \overline{I}))$
  with $IT(I_0) = ann\; \texttt{interface}\; I_0\; \texttt{extends}\; I_1 \ldots I_n \{\, \overline{meth}\,\}$ and $I \in \overline{I}$ if $I_i <: I, i \in 1..n$

The definition of mbody reconstructs the full set of supertypes $\overline{I}$ and then delegates the work to two other auxiliary functions: $\mathsf{needed}(m, \overline{I})$ and $\mathsf{override}(meth, \overline{meth})$.

needed recovers from the interface table only the "needed" methods, that is, the non-static ones that are not reachable by another, less specific superinterface. Since the second parameter of needed is a set, we can choose an arbitrary element to be I_0. In the definition we denote by $\mathsf{originalMethod}(m, I) = meth$ the non-static method called m defined directly in I. Formally:

- $\mathsf{originalMethod}(m, I_0) = meth$
  with $IT(I_0) = ann\; \texttt{interface}\; I_0\; \texttt{extends}\; \overline{I}\{\, \overline{meth}\,\}$, $meth \in \overline{meth}$ not static, $m = m^{meth}$
- $\mathsf{originalMethod}(m, I_0) \in \mathsf{needed}(m, I_0 \ldots I_n) =$
  $\nexists i \in 1..n$ such that $\mathsf{originalMethod}(m, I_i)$ is defined and $I_i <: I_0$

override models how a method in an interface can override implementations in its superinterfaces, even in the case of conflicts. Note how the special value None is used, and how (the 5th case) overriding can solve a conflict.

- $\mathsf{override}(\mathsf{None}, \emptyset) = \mathsf{None}$
- $\mathsf{override}(meth, \emptyset) = meth$
- $\mathsf{override}(\mathsf{None}, meth) = meth$
- $\mathsf{override}(\mathsf{None}, \overline{mh;}) = \mathsf{mostSpecific}(\overline{mh;})$
- $\mathsf{override}(meth, \overline{meth}) = meth$
  with $\forall meth' \in \overline{meth} : meth <: meth'$

The definition mostSpecific returns the most specific method whose type is the subtype of all the others. Since method subtyping is a partial ordering, mostSpecific may not be defined, this in turn forces us to rely on the last clause of override; otherwise the whole mbody would not be defined for that specific m. Rule (T-INTF) relies on this behavior.

- $\mathsf{mostSpecific}(\overline{meth}) = meth$
  with $meth \in \overline{meth}$ and $\forall meth' \in \overline{meth} : meth <: meth'$

To illustrate the mechanism of mbody, we present an example. We compute $\mathsf{mbody}(\texttt{m}, \texttt{D})$:

```
interface A { Object m(); }
interface B extends A { default Object m() {return
    this.m();} }
interface C extends A {}
interface D extends B, C { String m(); }
```

- First {A,B,C}, the full set of supertypes of D is obtained.
- Then we compute needed(m, {A,B,C}) = `default Object m(){...}`, that is B.m. That is, we do not consider either C.m (since m is not declared directly in C, hence originalMethod(m, C) is undefined) or B.m (that is a subtype of A, thus B.m hides A.m).
- The final step computes override(D.m, B.m) = D.m, by the last case of override we get that D.m hides B.m successfully (String is a subtype of Object). Finally we get mbody(m, D) = D.m.

B. What @Obj Generates

This section presents a formal definition for most of the generated methods by @Obj.

B.1 Translation

The translation functions of @Obj and @ObjOf are presented in Figure 6. Note that it is necessary to explicitly check if the interface is valid for annotation:

- $\mathsf{valid}(I_0) = \forall m \in \mathsf{dom}(I_0)$, if $mh; = \mathsf{mbody}(m, I_0)$, one of the following cases is satisfied: $\mathsf{isField}(meth), \mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$
- $\mathsf{isField}(I\,m\texttt{();}) = \text{not } \mathsf{special}(m)$
- $\mathsf{isWith}(I'\ \texttt{with\#}m(I\,x)\texttt{;}, I_0) =$ $I_0 <: I', \mathsf{mbody}(m, I_0) = I\,m\texttt{();}$ and not $\mathsf{special}(m)$
- $\mathsf{isSetter}(I'\ \texttt{\_}m(I\,x)\texttt{;}, I_0) =$ $I_0 <: I', \mathsf{mbody}(m, I_0) = I\,m\texttt{();}$ and not $\mathsf{special}(m)$

That is, we can categorize all abstract methods in a pattern that we know how to implement: it is either a field getter, a with method or a setter.

Moreover, we check that the method of is not already defined by the user. In the formalization an existing definition of the of method is an error. However, in the prototype (which also needs to account for overloading), the check is more complex as it just checks that an of method with the same signature of the one being generated is not already present.

We write with#*m* to append *m* to with, following the camelCase rule. The first letter of *m* must be lower-case and is changed to upper-case upon appending. For example with#foo=withFoo. Special names $\mathsf{special}(m)$ are with and all identifiers of the form with#*m*.

The refine function: $\mathsf{refine}(I_0, \overline{meth})$ is defined as follows:

- $I_0\ \texttt{with\#}m(I\ \texttt{\_val);} \in \mathsf{refine}(I_0, \overline{meth}) =$ $\mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m, I_0), I_0)$, $\texttt{with\#}m \notin \mathsf{dom}(\overline{meth})$
- $I_0\ \texttt{\_}m(I\ \texttt{\_val);} \in \mathsf{refine}(I_0, \overline{meth}) =$ $\mathsf{isSetter}(\mathsf{mbody}(\texttt{\_}m, I_0), I_0)$, $\texttt{\_}m \notin \mathsf{dom}(\overline{meth})$

The methods generated in the interface are with- and setters. The methods are generated when they are unimplemented in I_0, because the return types need to be refined. To determine whether the methods need to be generated, we check if such with- or setter methods are required by I_0, but not declared directly in I_0.

The ofMethod function: The function ofMethod generates the method of, as an object factory. To avoid boring digressions into well-known ways to find unique names, we assume that all methods with no parameters do not start with an underscore, and we prefix method names with underscores to obtain valid parameter names for of.

- $\mathsf{ofMethod}(I_0) =$ `static` I_0 `of(`I_1 `_`m_1`,` $\ldots$ I_n `_`m_n`){`
  `return new` I_0`(){`
  $I_1\ m_1 =$ `_`m_1`;` $\ldots$ $I_n\ m_n =$ `_`m_n`;`
  $I_1\ m_1$`(){return` m_1`;}` $\ldots$ $I_n\ m_n$`(){return` m_n`;}`
  $\mathsf{withMethod}(I_1, m_1, I_0, \overline{e}_1) \ldots \mathsf{withMethod}(I_n, m_n, I_0, \overline{e}_n)$
  $\mathsf{setterMethod}(I_1, m_1, I_0) \ldots \mathsf{setterMethod}(I_n, m_n, I_0)$
  `};}`
  with $I_1\ m_1\texttt{();}, \ldots I_n\ m_n\texttt{();} = \mathsf{fields}(I_0)$
  and $\overline{e}_i = m_1\texttt{,} \ldots \texttt{,}m_{i-1}\texttt{,\_val,}m_{i+1}\texttt{,} \ldots \texttt{,}m_n$

Note that, the function $\mathsf{fields}(I_0)$ denotes all the fields in the current interface:

- $meth \in \mathsf{fields}(I_0) =$ $\mathsf{isField}(meth)$ and $meth = \mathsf{mbody}(m^{meth}, I_0)$

For methods inside the interface with the form $I_i\ m_i\texttt{();}$

- m_i is the field name, and has type I_i.
- m_i() is the getter and just returns the current field value.
- if a method with#$m_i()$ is required, then it is implemented by calling the of method using the current value for all the fields except for m_i. Such new value is provided as a parameter. This corresponds to the expressions $\overline{e}_i$.
- _m_i(I_i _val) is the setter. In our prototype we use name m_i, here we use the underscore to avoid modeling overloading.

The auxiliary functions are defined below. Note that we do not need to check if some header is a subtype of what we would generate, this is ensured by $\mathsf{valid}(I_0)$.

- $\mathsf{withMethod}(I, m, I_0, \overline{e}) =$ I_0 `with#`m`(`I `_val){return` I_0`.of(`$\overline{e}$`);}` with $\mathsf{mbody}(\texttt{with\#}m, I_0)$ having the form $mh;$
- $\mathsf{withMethod}(I, m, I_0, \overline{e}) = \emptyset$ otherwise
- $\mathsf{setterMethod}(I, m, I_0) =$ I_0 `_`m`(`I `_val){`m`= _val;return this;}` with $\mathsf{mbody}(\texttt{\_}m, I_0)$ having the form $mh;$
- $\mathsf{setterMethod}(I, m, I_0) = \emptyset$ otherwise

B.2 Other Features

We do not formally model non-fluent setters and the with method. An informal explanation of how those methods are generated is given next:

- For methods inside the interface with the form `void` $m(I\,x)$`;`:
  - Check if method $I\,m\texttt{();}$ exists. If not, generate error (that is, $\mathsf{valid}(I_0)$ is false).

- Generate the implemented setter method inside `of`:
  `public void` m`(`I `_val){` m`=_val;}`
  There is no need to refine the return type for non-fluent setters, thus we do not need to generate the method header in the interface body itself.

- For methods with the form I' `with(`I x`);`:
  - I must be an interface type (no classes or primitive types).
  - As before, check that I' is a supertype of the current interface type I_0.
  - Generate implemented `with` method inside `of`:
    `public` I_0 `with(`I `_val){`
    `if(_val instanceof` I_0`){return (`I_0`)_val;}`
    `return` I_0`.of(`$e_1 \dots e_n$`);}`
    with e_i =`_val.`m_i`()` if I has a m_i`()` method where $m_1 \dots m_n$ are fields of I_0; otherwise $e_i = m_i$.
  - If needed, as for `with-` and setters, generate the method headers with refined return types in the interface.

C. Lemmas and Theorems

C.1 LEMMA 1 and Proof

Lemma 1 (a)**.** *For any expression e under an interface table $\mathcal{I}$ IT where $\Gamma \vdash e \in I^{\mathcal{I}}$, $\mathcal{I}$ has `@ObjOf` annotation and $[\![\mathcal{I}]\!] = \mathcal{I}'$, then under the interface table $\mathcal{I}'$ IT, $\Gamma \vdash e \in I^{\mathcal{I}}$.*

Proof. By induction on the typing rules: by the grammar shown in Figure 10, there are 6 cases for an arbitrary expression e:

- Variables are typed in the same exact way.
- Field update. The type preservation is ensured by induction.
- A method call (normal, static or super). The corresponding method declaration won't be "removed" by the translation, also the types remain unchanged. The only work `@ObjOf` does is adding a static method `of` to the interface, however, a pre-condition of the translation is `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, so adding `of` method has no way to affect any formerly well typed method call.
- An object creation. Adding the `of` method doesn't introduce unimplemented methods to an interface, moreover, the static method is not inheritable, hence after translation such an object creation still type checks and has the right type by induction.

□

Lemma 1 (b)**.** *For any expression e under an interface table $\mathcal{I}$ IT where there is no heir of $I^{\mathcal{I}}$, $\Gamma \vdash e \in I^{\mathcal{I}}$, $\mathcal{I}$ has `@Obj` annotation and $[\![\mathcal{I}]\!] = \mathcal{I}'$, then under the interface table $\mathcal{I}'$ IT, $\Gamma \vdash e \in \_ <: I^{\mathcal{I}}$.*

Proof. The proof follows the same scheme of Lemma 1 (a), but for the case of method call the return type may be refined with a subtype. This is still ok since we require $\_ <: I^{\mathcal{I}}$. On the other side, this weaker result still allows the application on the method call typing rules, since in the premises the types of the actual parameter are required to be a subtype of the formal one. □

C.2 LEMMA 2 and Proof

Lemma 2 (a)**.** *If $\mathcal{I}$ has `@ObjOf` annotation and $I^{\mathcal{I}}$ OK in $\mathcal{I}$ IT, then $[\![\mathcal{I}]\!]$ OK in $[\![\mathcal{I}]\!]$ IT.*

Proof. By the rule (T-INTF) in Figure 11, we divide the proof into two parts.

Part I. For each default or static method in the domain of $[\![I^{\mathcal{I}}]\!]$, the type of the return value is compatible with the method's return type.

Since $\mathcal{I}$ OK, and by Lemma 1 (a), all the existing default and static methods are well typed in $[\![\mathcal{I}]\!]$, except for the new method `of`. It suffices to prove that it still holds for $\mathsf{ofMethod}(I)$.

By the definition of $\mathsf{ofMethod}(I)$, the return value is an object

`return new` $I^{\mathcal{I}}$`(){ ... }`

To prove it is of type $I^{\mathcal{I}}$, we use the typing rule (T-OBJ).

- All field initializations are type correct. By the definition of $\mathsf{ofMethod}(I^{\mathcal{I}})$ in Appendix B.1, the fields $m_1, \dots, m_n$ are initialized by `of`'s arguments, and types are compatible.
- All method bodies are well-typed.
  - Typing of the i-th getter m_i.

    $$\Gamma, m_i : I_i, \texttt{this} : I^{\mathcal{I}} \vdash m_i \in I_i$$

    We know that $I_i = I^{mh_i}$ since the i-th getter has its return type the same as the corresponding field m_i.
  - Typing of the `with-` method of an arbitrary field m_i. By Appendix B.1, if the `with-` method of m_i is well-defined, it has the form

    $$I^{\mathcal{I}}\ \texttt{with\#}m_i\texttt{(}I_i\ \texttt{\_val)\{ return}\ I^{\mathcal{I}}\texttt{.of(}\overline{e}_i\texttt{);\}}$$

    $\overline{e}_i$ is obtained by replacing m_i with `_val` in the list of fields, and since they have the same type I_i, the arguments $\overline{e}_i$ are compatible with $I^{\mathcal{I}}$`.of` method. Hence

    $$\Gamma, m_1 : I_1 \dots m_n : I_n, \texttt{this} : I^{\mathcal{I}}, \texttt{\_val} : I_i \vdash I^{\mathcal{I}}\texttt{.of(}\overline{e}_i\texttt{)} \in I^{\mathcal{I}}$$

    We know that $I^{\mathcal{I}} = I^{mh_i}$ by the return type of `with#`m_i shown as above.
  - Typing of the i-th setter `_`m_i. If the `_`m_i method is well-defined, it has the form

    $$I^{\mathcal{I}}\ \texttt{\_}m_i\texttt{(}I_i\ \texttt{\_val)\{}\ m_i\texttt{= \_val;return this;\}}$$

    By (T-UPDATE), the assignment "m_i`= _val;`" is correct since m_i and `_val` have the same type I_i, and the return type is decided by `this`.

    $$\Gamma, \texttt{this} : I^{\mathcal{I}}, \texttt{\_val} : I_i \vdash \texttt{this} \in I^{\mathcal{I}}$$

    We know that $I^{\mathcal{I}} = I^{mh_i}$ by the return type of `_`m_i shown as above.

- All method headers are valid with respect to the domain of $I^{\mathcal{I}}$. Namely

  $$\mathsf{sigvalid}(mh_1 \ldots mh_n, I)$$

  For convenience, we use "*meth* in $\mathsf{ofMethod}(I^{\mathcal{I}})$" to denote that *meth* is one of the implemented methods in the return expression of $\mathsf{ofMethod}(I^{\mathcal{I}})$, namely `new` $I^{\mathcal{I}}$`()``{...}`.
  - For the i-th getter m_i,

    $$\begin{array}{rl} & I_i\ m_i\texttt{()\{...\}}\ \text{in}\ \mathsf{ofMethod}(I^{\mathcal{I}}) \\ \text{implies} & I_i\ m_i\texttt{();} \in \mathsf{fields}(I^{\mathcal{I}}) \\ \text{implies} & I_i\ m_i\texttt{();} = \mathsf{mbody}(m_i, I^{\mathcal{I}}) \\ \text{implies} & I_i\ m_i\texttt{();} <: \mathsf{mbody}(m_i, I^{\mathcal{I}}) \end{array}$$

  - For the `with#`m_i method,

    $$\begin{array}{rl} & I^{\mathcal{I}}\ \texttt{with\#}m_i\texttt{(}I_i\ \texttt{\_val)\{...\}}\ \text{in}\ \mathsf{ofMethod}(I^{\mathcal{I}}) \\ \text{implies} & \mathsf{mbody}(\texttt{with\#}m_i, I^{\mathcal{I}})\ \text{is of form}\ mh\texttt{;} \\ \text{with} & \mathsf{valid}(I^{\mathcal{I}}) \\ \text{implies} & \mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m_i, I^{\mathcal{I}}), I^{\mathcal{I}}) \\ \text{implies} & I^{\mathcal{I}}\ \texttt{with\#}m_i\texttt{(}I_i\ \texttt{\_val);} <: \mathsf{mbody}(\texttt{with\#}m_i, I^{\mathcal{I}}) \end{array}$$

  - For the i-th setter $\_m_i$,

    $$\begin{array}{rl} & I^{\mathcal{I}}\ \_m_i\texttt{(}I_i\ \texttt{\_val)\{...\}}\ \text{in}\ \mathsf{ofMethod}(I^{\mathcal{I}}) \\ \text{implies} & \mathsf{mbody}(\_m_i, I^{\mathcal{I}})\ \text{is of form}\ mh\texttt{;} \\ \text{with} & \mathsf{valid}(I^{\mathcal{I}}) \\ \text{implies} & \mathsf{isSetter}(\mathsf{mbody}(\_m_i, I^{\mathcal{I}}), I^{\mathcal{I}}) \\ \text{implies} & I^{\mathcal{I}}\ \_m_i\texttt{(}I_i\ \texttt{\_val);} <: \mathsf{mbody}(\_m_i, I^{\mathcal{I}}) \end{array}$$

- All abstract methods in the domain of $I^{\mathcal{I}}$ have been implemented. Namely

  $$\mathsf{alldefined}(mh_1 \ldots mh_n, I)$$

  Here we simply refer to $\mathsf{valid}(I^{\mathcal{I}})$, since it guarantees each abstract method to satisfy isField, isWith or isSetter. But that object includes all implementations for those cases. A getter m_i is generated if it satisfies isField; a `with-` method is generated for the case isWith, by the definition of withMethod; a setter for isSetter, similarly, by the definition of setterMethod. Hence it is of type $I^{\mathcal{I}}$ by (T-OBJ).

Part II. Next we check that in $[\![\mathcal{I}]\!]$,

$$\mathsf{dom}([\![\mathcal{I}]\!]) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth}) \cup \mathsf{dom}(meth')$$

Since $\mathcal{I}$ OK, we have $\mathsf{dom}(\mathcal{I}) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$, and hence it is equivalent to prove

$$\mathsf{dom}([\![\mathcal{I}]\!]) = \mathsf{dom}(I^{\mathcal{I}}) \cup \mathsf{dom}(meth')$$

This is obvious since a pre-condition of the translation is `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, so $meth'$ doesn't overlap with $\mathsf{dom}(I^{\mathcal{I}})$. The definition of dom is based on mbody, and here the new domain $\mathsf{dom}([\![\mathcal{I}]\!])$ is only an extension to $\mathsf{dom}(I)$ with the `of` method, namely $meth'$. Also note that after translation, there are still no methods with conflicted names, since the `of` method was previously not in the domain, hence $[\![\mathcal{I}]\!]$ is well-formed, which finishes our proof. □

Lemma 2 (b)**.** *If $\mathcal{I}$ has `@Obj` annotation $I^{\mathcal{I}}$ OK in $\mathcal{I}$ IT and there is no heir of $I^{\mathcal{I}}$, then $[\![\mathcal{I}]\!]$ OK in $[\![\mathcal{I}]\!]$ IT.*

Proof. **Part I.** Similarly to what already argued for Lemma 2 (a), since $\mathcal{I}$ OK, and by Lemma 1 (b), all the existing default and static methods are well typed in $[\![\mathcal{I}]\!]$ IT. The translation function delegates its work to `@ObjOf` in such way that we can refer to Lemma 2 (a) to complete this part. Note that all the methods added (directly) by `@Obj` are abstract, and thus there is no body to typecheck.

Part II. Similar to what we already argued for Lemma 2 (a), but we need to notice that the newly added methods are valid refinements for already present methods in $\mathsf{dom}(I^{\mathcal{I}})$ before the translation. Thus by last clause of the definition of override(_), mbody(_) is defined on the same method names. □

C.3 THEOREM and Proof

Theorem 1 (@ObjOf tuning)**.** *If a given interface table $\mathcal{I}$ IT is OK where $\mathcal{I}$ has `@ObjOf`, $\mathsf{valid}(I^{\mathcal{I}})$ and `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, then the interface table $[\![\mathcal{I}]\!]$ IT is OK.*

Proof. Lemma 2 (a) already proves that $[\![\mathcal{I}]\!]$ is OK. On the other hand, for any $\mathcal{I}' \in \text{IT}\backslash\mathcal{I}$, by Lemma 1 (a), we know that all its methods are still well-typed, and the generated code in translation of `@ObjOf` is only a static method `of`, which has no way to affect the domain of $\mathcal{I}'$, so after translation rule (T-INTF) can still be applied, which finishes our proof. □

Theorem 2 (@Obj tuning)**.** *If a given interface table $\mathcal{I}$ IT is OK where $\mathcal{I}$ has `@Obj`, $\mathsf{valid}(I^{\mathcal{I}})$ and `of` $\notin \mathsf{dom}(I^{\mathcal{I}})$, and there is no heir of $I^{\mathcal{I}}$, then the interface table $[\![\mathcal{I}]\!]$ IT is OK.*

Proof. Similar to what already argued for Theorem 1, we can apply Lemma 2 (b) and Lemma 1 (b). Then we finish by Theorem 1. □