Classless Java: Tuning Java Interfaces

Abstract

Java 8 introduced *default methods*, allowing interfaces to have method implementations. When combined with (multiple) interface inheritance, default methods provide a basic form of multiple inheritance. However, using this combination to simulate more advanced forms of multiple inheritance quickly becomes cumbersome, and appears to be quite restricted.

This paper shows that, with a simple language feature, default methods and interface inheritance are in fact very expressive. Our proposed language feature, called *object interfaces*, enables powerful object-oriented idioms, using multiple inheritance, to be expressed conveniently in Java. Object interfaces refine conventional Java interfaces in three different ways. Firstly, object interfaces have their own object instantiation mechanism, providing an alternative to class constructors. Secondly, object interfaces support *abstract state operations*, providing a way to use multiple inheritance with state in Java. Finally, object interfaces allow type refinements that are often tricky to model in conventional class-based approaches. Interestingly, object interfaces do not require changes to the runtime, and they also do not introduce any new syntax: all three features are achieved by reinterpreting existing Java syntax, and are translated into regular Java code without loss of type-safety. Since no new syntax is introduced, it would be incorrect to call object interfaces a language extension or syntactic sugar. So we use the term *language tuning* to characterize this kind of language feature. An implementation of object interfaces using Java annotations and a formalization of the static and dynamic semantics are presented. Moreover, the usefulness of object interfaces is illustrated through various examples.

1 Introduction

Java 8 introduced *default methods*, allowing interfaces to have method implementations. The main motivation behind the introduction of default methods in Java 8 is *interface evolution*. That is, to allow interfaces to be extended over time while preserving backwards compatibility. It soon became clear that default methods could also be used to emulate something similar to *traits* [22]. The original notion of traits by Scharli et al. prescribes, among other things, that: 1) a trait provides a set of methods that implement behavior; and 2) a trait does not specify any state variables, so the methods provided by traits do not access state variables directly. Java 8 interfaces follow similar principles too. Indeed, a detailed description of how to emulate trait-oriented programming in Java 8 can be found in the work by Bono et al. [4]. The Java 8 team designing default methods, was also fully aware of that secondary use of interfaces, but it was not their objective to model traits: “The key goal of adding default methods to Java was "interface evolution", not "poor man’s traits"” [11]. As a result, they were happy to support the secondary use of interfaces with default methods as long as it did not make the implementation and language more complex.

Still, the design is quite conservative and appears to be quite limited in its current form to model advanced forms of multiple inheritance. Indeed, our own personal experience of combining default methods and multiple interface inheritance in Java to achieve multiple implementation inheritance is that many workarounds and boilerplate code are needed. In particular, we encountered difficulties because:

- *Interfaces have no constructors.* As a result, classes are still required to create objects, leading to substantial boilerplate code for initialization.
- *Interfaces do not have state.* This creates a tension between using multiple inheritance and having state. Using setter and getter methods is a way out of this tension, but this

workaround requires tedious boilerplate classes that later implement those methods.

- *Useful, general purpose methods require special care in the presence of subtyping.* Methods such as *fluent* setters [9], not only require access to the internal state of an object, but they also require their return types to be refined in subtypes.

Clearly, a way around those difficulties would be to change Java and just remove these limitations. Scala's own notion of traits [17], for example, allows state in traits. Of course adding state (and other features) to interfaces would complicate the language and require changes to the compiler, and this would go beyond the goals of Java 8 development team.

This paper takes a different approach. Rather than trying to get around the difficulties by changing the language in fundamental ways, we show that, with a simple language feature, default methods and interface inheritance are in fact very expressive. Our proposed language feature enables powerful object-oriented idioms, using multiple inheritance. We call the language feature *object interfaces*, because such interfaces can be instantiated directly, without the need for an explicit class definition. Moreover, object interfaces support *abstract state operations*, providing a way to use multiple inheritance with state in Java. The abstract state operations include various common utility methods (such as getters and setters, or clone-like methods). In the presence of subtyping, such 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.

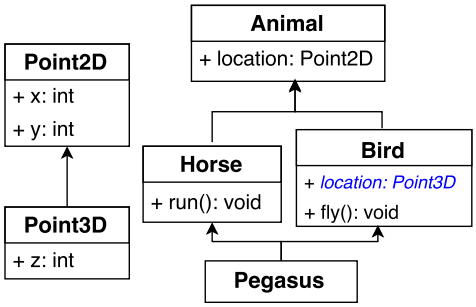
Object interfaces do not require changes to the Java runtime or compiler, and they also do not introduce any new syntax. All three features of object interfaces are achieved by reinterpreting existing Java syntax, and are translated into regular Java code without loss of type-safety. Since no new syntax is introduced, it would be incorrect to call object interfaces a language extension or syntactic sugar. So we use the term *language tuning* instead. Language tuning sits in between a lightweight language extension and a glorified library. Language tuning can offer many features usually implemented by a real language extension, but because it does not modify the language syntax pre-existing tools can work transparently on the tuned language. To exploit the full benefits of language tuning, our prototype implementation of object interfaces uses Java annotations to do AST rewriting, allowing existing Java tools (such as IDEs) to work out-of-the-box with our implementation. As a result, we could experiment object interfaces with several interesting Java programs, and conduct various case studies.

To formalize object interfaces, we propose Classless Java (CJ): a FeatherweightJava-style [13] calculus, which captures the essence of interfaces with default methods. The semantics of object interfaces is given as a syntax-directed translation from CJ to itself. In the resulting CJ code, all object interfaces are translated into regular CJ (and Java) interfaces with default methods. The translation is proved to be type-safe, ensuring that the translation does not introduce type-errors in client code.

To evaluate the usefulness of object interfaces, we illustrate 3 applications. The first application is a simple solution to the Expression Problem [25], supporting independent extensibility [27], and without boilerplate code. The second application shows how embedded DSLs using fluent interfaces [9] can be easily defined using object interfaces. The last application is a larger case study for a simple Maze game implemented with multiple inheritance. For the last application we show that there is a significant reduction in the numbers of lines of code when compared to an existing implementation [4] using plain Java 8. Noteworthy, all applications are implemented without defining a single class!

In summary, the contributions of this paper are:

- **Object Interfaces:** A simple feature that allows various powerful multiple-inheritance




```
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).

programming idioms to be expressed conveniently in Java.

- **Classless Java (CJ):** A simple formal calculus that models the essential features of Java 8 interfaces with default methods, and can be used to formally define the translation of object interfaces. We prove several properties of the translation[1].
- **Implementation and Case Studies:** We have a prototype implementation of object interfaces, using Java annotations and AST rewriting. Moreover, the usefulness of object interfaces is illustrated through various examples and case studies.
- **Language Tuning:** We identify the concept of language tuning and describe object interfaces as an example. We also discuss how other existing approaches, such as the annotations in project Lombok [29], can be viewed as language tuning.

2 A Running Example: Animals

This section illustrates how object interfaces, expressed using the `@Obj` annotation, enable powerful idioms using multiple inheritance in Java. To propose a standard example for multiple inheritance, we show `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. 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. 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 modeling 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 behavior from

[1] Proofs and prototype implementation are available in the supplementary materials.
[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.

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 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 a `@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, which still does not appear to justify the `@Obj` annotation. Now moving on to modeling the complete animal system where `Animal`s will include a `location` representing

their positions in space, we shall see that modeling stateful components using plain Java 8 quickly becomes cumbersome. The `@Obj` annotation comes to rescue here and avoids significant amounts of boilerplate code.

`Point2D`: simple immutable data with fields 2D points should keep track of their coordinates. The usual approach to model points in Java would be to use a class with fields for the coordinates. However here we will illustrate how points are modeled with interfaces. Since Java disallows fields inside interfaces, state is modeled using abstract (getter) methods:

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

Unfortunately, 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 our `@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 are the fields, 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 in our code generation as `with-` methods. For example:

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

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){ ... } default Point2D withY(int val){ ... } }
```

is expanded into

	Example	Description
"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`.

Figure 2 Abstract state operations, for a field `x`, allowed by the `@Obj` annotation.

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

Only code for methods needing implementation is generated. Thus, programmers can easily customize the behavior for their special needs.

`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 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.

Summary Dealing with state (mutable or not) in object interfaces relies on a notion of abstract state, where only *methods* that interact with state are available to users. Object interfaces provide support for four different types of abstract state operations, which are summarized in Figure 2. The 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 7.2.

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 usually poses a challenge 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 semantic 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 property updater.

`Point3D` and properties 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) properties 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`.

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 3. Writing such code by hand is error-prone. For example a distracted programmer may swap the arguments of calls to

```
interface Point3D extends Point2D {
   Point3D withX(int val); Point3D withY(int val); Point3D withZ(int val);
   Point3D with(Point2D 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 3 Generated boilerplate code.

`Point3D.of`. Note how `with-` methods are automatically refined in their return type, 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`**.

2.5 Advanced Multiple Inheritance with Object Interfaces

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 Interaction of Interface Methods with Interface Composition

Before formalizing Classless Java and object interfaces, it is helpful to informally discuss the behavior of methods in Java 8 interfaces. In particular it is useful to understand how Java 8 interfaces differ from conventional trait models.

3.1 Methods in Java 8 Interfaces

In Java 8 interfaces there are three types of methods: abstract, default, and static methods. Default and static methods were not allowed in interfaces in previous versions of Java.

Static methods are handled in a very clean way: they are visible only in the interface in which they are explicitly defined. This means the following code is ill-typed.

```
interface A0 { static int m(){return 1;} }
interface B0 extends A0 {}
... B0.m()//ill typed
```

This is different from the way static methods are handled in classes. Here static methods have simply no interaction with interface composition (**`extends`** or **`implements`**).

Abstract method composition is accepted when there exists a most specific one. For example, here method `Integer m()` from `B1` is visible in `C1`.

```
interface A1 { Object m(); }
interface B1 { Integer m();}
interface C1 extends A1,B1 {} //accepted
```

Default methods conflict with any other default or abstract method. For example the following code is rejected due to method conflicts.

```
interface A2 { default int m() {return 1;}}
interface B2 { int m(); }
interface C2 { default int m() {return 2;}}
interface D2 extends A2,B2 {} //rejected due to conflicting methods
interface E2 extends A2,C2 {} //rejected due to conflicting methods
```

Note how this is different from what happens in most trait models, where `D2` would be accepted, and the implementation in `A2` would be part of the behavior of `D2`.

Resolving conflicts: A method in the current interface wins over any method in its super-interfaces, provided that the method is the most specific one. This method also overrides conflict due to inheritance. For example, the following code is accepted, but would be rejected (see before) if the method `m` was not redefined in `D3` and `E3`.

```
interface D3 extends A2,B2 { int m(); } //accepted
interface E3 extends A2,C2 { default int m(){return 42;} } //accepted
```

3.2 Classifying Outcomes of Interface Composition

We now try to classify possible outcomes for composition of methods with the same name (and signature). We will use the following (correct) declarations:

```
interface A1 { T m(); }
interface A2 extends A1 { default T m(){ ... } }
interface A3 extends A2 { T m(); }

interface B1 { default T m(){ ... } }
interface B2 extends B1 { T m(); }
interface B3 extends B2 { default T m(){ ... } }
```

What happens if a new interface `M` extends one $\texttt{A}_i$ and one $\texttt{B}_j$?

M extends	**A1**	**A2**	**A3**
B1	conservative error	conflict error	conservative error
B2	both abstract (accepted)	conservative error	both abstract (accepted)
B3	**conservative error**	conflict error	conservative error

- **conflict error** happens when the methods from both interfaces are implemented, which is also an error in most trait models.
- **both abstract (accepted)** happens when the methods from both interfaces are abstract, which is also considered correct in all trait models.
- **conservative error** happens when only one method is implemented (leaving another one abstract), which is different from what we would expect in a trait model, but is coherent with the conservative idea that a method defined in an interface should not silently satisfy a method in another one.

A bug: During our experimentation, we found a bug in ECJ (Eclipse compiler for Java): the case **M** extending **B3** and **A1** is accepted by ECJ4.5.1 and rejected by javac. By email communication with Brian Goetz (leading Java 8 designer) we have confirmed that the

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

Figure 4 Grammar of Classless Java

expected behavior is rejection, hence this is a bug in ECJ. This bug was also reported by others and is fixed in the ECJ developer branch, but not released as a stable version yet.

4 Formal Semantics

This section presents a formalization of Classless Java: a minimal FeatherweightJava-like calculus which models the essence of Java interfaces with default methods. This formalization is used in Section 5 to define the semantics of object interfaces.

4.1 Syntax

Figure 4 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 ($I.\texttt{super}.m(\overline{e})$), 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 ($x\texttt{=}e\texttt{;}e'$) 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 $\texttt{return}\ x\texttt{=}e\texttt{;}e'$ to represent $x\texttt{=}e\texttt{;return}\ e'$ 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.

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

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

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

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

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

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

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

Figure 5 CJ Typing

4.2 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 Section 4.3, 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 5 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:

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

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.

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

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:

- $\mathsf{alldefined}(mh_1 \ldots mh_n, I) \quad = \quad \forall m$ 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 $\mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$. 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 $\mathsf{dom}(I_1) \ldots \mathsf{dom}(I_n)$ or if there are methods in any $\mathsf{dom}(I_i)$ and $\mathsf{dom}(I_j)$ $(i, j \in 1..n)$ conflict.

4.3 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.

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

The definition of mbody reconstructs the full set of supertypes $\bar{I}$ and then delegates the work to two other auxiliary functions: $\mathsf{needed}(m, \bar{I})$ 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 $IT(I_0) = \ ann\ \texttt{interface}\ I_0\ \texttt{extends}\ \bar{I}\ \texttt{\{}\ \overline{meth}\ \texttt{\}}, meth \in \overline{meth}$ not static, $m = m^{meth}$
- $\mathsf{originalMethod}(m, I_0) \in \mathsf{needed}(m, I_0 \ldots I_n) \quad =$
  $\nexists i \in 1..n$ 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.

- $\mathsf{override}(\mathsf{None}, \emptyset) \quad = \quad \mathsf{None}$
- $\mathsf{override}(meth, \emptyset) \quad = \quad meth$
- $\mathsf{override}(\mathsf{None}, meth) \quad = \quad meth$
- $\mathsf{override}(\mathsf{None}, \overline{mh;}) \quad = \quad \mathsf{mostSpecific}(\overline{mh;})$
- $\mathsf{override}(meth, \overline{meth}) \quad = \quad meth$
  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 $\mathsf{needed}(\texttt{m}, \{\texttt{A},\texttt{B},\texttt{C}\}) = \texttt{default Object m()\{...\}}$, that is B.m. That is, we do not consider either C.m (since m is not declared directly in C, hence $\mathsf{originalMethod}(\texttt{m}, \texttt{C})$ is undefined) or B.m (that is a subtype of A, thus B.m hides A.m).
- The final step computes $\mathsf{override}(\texttt{D.m}, \texttt{B.m}) = \texttt{D.m}$, by the last case of override we get that D.m hides B.m successfully (String is a subtype of Object). Finally we get $\mathsf{mbody}(\texttt{m}, \texttt{D}) = \texttt{D.m}$.

5 What @Obj Generates

This section shows what the @Obj annotation generates and presents a formal definition for most of the generated methods. 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.

5.1 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.

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

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

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 $I_1\ m_1$`();`, $\ldots$ $I_n\ m_n$`();` = $\mathsf{fields}(I_0)$ and $\overline{e}_i = m_1$`,` $\ldots$ `,`m_{i-1}`,_val,`m_{i+1}`,` $\ldots$ `,`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 $I_i\ m_i$`();`

- 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 $\mathsf{mbody}(\_m, I_0)$ having the form mh`;`
- $\mathsf{setterMethod}(I, m, I_0)$ = $\emptyset$ otherwise

5.2 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 $I\ m$`();` 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.

5.3 Guarantees

To understand to what extent our approach is correct (and more in general, what it means to say that a language tuning is correct) we identify three types of guarantees:

- *Self coherence*: the generated code itself is well-typed; that is, type errors are not present in code the user has not written. 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. We guarantee *Self coherence*.
- *Client coherence*: all the client code (for example method calls) that is well-typed before code generation is also well-typed after the generation. The annotation just adds more behavior without removing any functionality. We guarantee *Client coherence*.
- *Heir coherence*: interfaces (and in general classes) inheriting the instrumented code are well-typed if they were well-typed without the instrumentation. This would forbid adding any (default or abstract) method to the annotated interfaces, including type refinement. `@Obj` does not guarantee *Heir coherence*. 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 would be ill-typed after the translation.

This means that 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.

5.4 Results

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 $I^{\mathcal{I}}$ the name of an interface.
- An interface table IT is OK if under such interface table, all interfaces are OK.
- 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 instantiate it.

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

▶ **Theorem 2** (@Obj tuning). *If a given interface table $\mathcal{I}$ IT is OK where $\mathcal{I}$ has* `@Obj`*, $valid(I^{\mathcal{I}})$ and* `of` $\notin dom(I^{\mathcal{I}})$*, 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 typechecks before the translation is applied, if the annotated type has no subtypes and no objects of that type are created, then type safety 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` tuning theorem. `@Obj` tuning is more interesting: since `@Obj` does not guarantee heir coherence, we explicitly exclude the presence of heirs. In this way `@Obj` tuning guarantees only self and client coherence. The formal theorem proofs are available in the supplementary materials.

6 Implementation

Our implementation is based on an extension of Lombok. The Lombok project [29] is a Java library that aims at removing (or reducing) Java boilerplate code via annotations. There are a number of annotations provided by the original Lombok, including `@Getter`, `@Setter`, `@ToString` for generating getters, setters and `toString` methods, respectively. Furthermore, Lombok provides a number of interfaces for users to create custom transformations, as extensions to the original framework. A transformation is based on a handler, which acts on the AST for the annotated node and returns a modified AST for analysis and generation afterwards. Such a handler can either be a Javac handler or an Eclipse handler.

The annotation we created is `@Obj`. In Eclipse, with an interface annotated by `@Obj`, the automatic annotation 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. The custom transformation is easy and convenient to use. For example, this means that the IDE functionality for content assist and autocomplete will work for the newly generated methods. The biggest reasons to use Lombok rather than using a conventional Java annotation processor are:

- Lombok modifies 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 is capable of generating code *inside* a class/interface, which conventional Java annotation processors do not support.

Limitations Our prototype implementation using Lombok has certain limitations:

- The prototype does not support separate compilation yet. Currently all related interfaces have to appear in a single Java file. Therefore, changes to a single interface would require re-compiling the whole file. This compilation limitation is not caused by our algorithm. It is a Lombok implementation related issue: in Lombok it is hard to capture a type declaration from its reference, even harder when the type declaration is in other files (we have not found a way to do this yet).
- At this stage our implementation only realizes the Eclipse handler and our experiments are all conducted in Eclipse. The implementation for `javac` is missing.
- The current implementation does not take type-parameters into consideration, thus it does not support generics yet.

Comparison with other Lombok annotations The Lombok project provides a set of predefined annotations, including constructor generators similar as ours (e.g., `@NoArgsConstructor`, `@RequiredArgsConstructor` and `@AllArgsConstructor`). They generate various kinds of constructors for *classes*, with or without constructor arguments. This set of annotations is of great use, especially when used together with other features provided in Lombok (e.g., `@Data`). Moreover, the implementation of these annotations in Lombok gives us hints on how to implement `@Obj`. However, none of these annotations can model what we are doing with `@Obj`- generating constructor-methods (**of**) for *interfaces*. Apart from constructors, `@Obj` also provides other convenient features (including generating fluent setters, type refinement, etc), which the base Lombok project does not provide. Finally, while `@Obj` is formalized, none of Lombok's annotations have been studied in a formal way.

Lombok does language tuning We consider Lombok to be the most developed example of language tuning. While the authors of Lombok do not introduce a specific term for what they are doing, their slogan *"Spice up your java"* seems to be in line with the philosophy of language tuning. Some other examples of language tuning in Lombok include the `val` type, similar to `auto` in C# or C++04. Another library doing language tuning is CoFoJa [14], where annotations are used to insert pre-post conditions in generated bytecode.

7 Case Studies

In this section we conduct three case studies which reveal various advantages of using the `@Obj` annotation. The first case study provides a simple way to solve the Expression Problem while supporting multiple, independent extensions in Java. The second case study shows how to model an embedded DSL for SQL languages with fluent interfaces. Finally, the third case study models a simple game, and compares our implementation with an existing one, showing that the amount of code is reduced significantly using `@Obj`.

7.1 A Trivial Solution to the Expression Problem with Object Interfaces

The *Expression Problem* (EP) [25] is a well-known problem about modular extensibility issues in software evolution. Recently, a new solution [26] using only covariant type refinement was proposed. When this solution is modeled with interfaces and default methods, it can even provide independent extensibility [27]: the ability to assemble a system from multiple, independently developed extensions. Unfortunately, the required instantiation code makes a plain Java solution verbose and cumbersome to use. The `@Obj` annotation is enough to remove the boilerplate code, making the presented approach very appealing.

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 6, `Exp` is the common super-interface with operation `eval()` inside. Sub-interfaces `Lit` and `Add` extend interface `Exp` with default implementations for the `eval` operation. The number field `x` of a literal is represented as a getter method `x()` and expression fields (`e1` and `e2`) of an addition as getter methods `e1()` and `e2()`.

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

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

Adding a New Operation The difficulty of the EP in OO languages arises from adding

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

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

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

new operations. For example, adding a pretty printing operation would typically change all existing code. However, a solution should add operations in a type-safe and modular way. This turns out to be easily achieved with the assistance of `@Obj`. The code in Figure 6 (on the right) shows how to add the new operation `print`. Interface `ExpP` extends `Exp` with the extra method `print()`. Interfaces `LitP` and `AddP` are defined with default implementations of `print()`, extending base interfaces `Lit` and `Add`, respectively. Importantly, note that in `AddP`, the types 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 {}
@Obj interface LitPC extends ExpPC, LitP, LitC {}
@Obj interface AddPC extends ExpPC, AddP, AddC { ExpPC e1(); ExpPC e2(); }
```

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

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.

7.2 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 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");
```

7.3 A Maze Game

The last case study is a simplified variant of a Maze game, which is often used [10, 4] to evaluate code reuse ability related to inheritance and design patterns. In the game, there is a player with the goal of collecting as many coins as possible. She may enter a room with several doors to be chosen among. This is a good example because it involves code reuse (different kinds of doors inherit a common type, with different features and behavior), multiple inheritance (a special kind of door may require features from two other door types) and it also shows how to model operations `symmetric sum`, `override` and `alias` from trait-oriented programming. The game has been implemented using plain Java 8 and default methods by Bono et. al [4], and the code for that implementation is available online. We 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.

8 Related Work

In this section we discuss related work and how it compares to Classless Java.

Multiple Inheritance in Object Oriented Languages Many authors have argued in favor or against multiple inheritance. Multiple inheritance is very expressive, but difficult to model and implement, and can cause difficulty (including the famous diamond (fork-join) problem [5, 21], 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 [5], traits [22], and hybrid models such as CZ [16]. 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++ tries to provide a general solution to multiple inheritance by virtual inheritance, dealing with the diamond problem by keeping only one copy of the base class [7]. However C++ suffers from the object initialization problem [16]. It bypasses all constructor calls to virtual superclasses, which can potentially 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 a more restricted model than the C++ approach. Mixins allow to name 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 [28]. Scala traits are in fact more like linearized mixins. Scala avoids the object initialization problem by disallowing constructor parameters, causing no ambiguity in cases such as 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 [16] which aims to do multiple inheritance without the diamond problem. They divide inheritance into two separate concepts: inheritance dependency and implementation inheritance. Using a combination of `requires` and `extends`, a program with diamond inheritance can be 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. `@Obj` does not complicate the class structure, and state can also coexist with multiple inheritance.

Simplifying the mixins approach, traits [22] draw a strong line between units of reuse and object factories. Traits, as units of reusable code, contain only methods as reusable functionality, ignoring state and state initialization. Classes, as object factories, require functionality from (multiple) traits. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the **default** keyword) inside interfaces. The

introduction of default methods opens the gate for various flavors of multiple inheritance in Java. Traits offer an algebra of composition operations like sum, alias and exclusion, providing explicit conflict resolution. Former work [4] provides details on mimicking the trait algebra through Java 8 interfaces. We briefly recall the main points of their encoding; however we propose a different representation of **exclusion**. The first author of [4] agreed (via personal communication) that our revised version for exclusion is cleaner, typesafe and more direct.

- **Symmetric sum** can be obtained by simple multiple inheritance between interfaces.

```
interface A { int x(); } interface B { int y(); } interface C extends A, B {}
```

- **Overriding** a conflict is obtained by specifying which super interface take precedence.

```
interface A { default int m() {return 1;} }
interface B { default int m() {return 2;} }
interface C extends A, B { default int m() {return B.super.m();} }
```

- **Alias** is creating a new method delegating to the existing super interface.

```
interface A { default int m() {return 1;} }
interface B extends A { default int k() {return A.super.m();} }
```

- **Exclusion**: exclusion is also supported in Java, where method declarations can hide the default methods correspondingly in the super interfaces.

```
interface A { default int m() {return 1;} }
interface B extends A { int m(); }
```

There are also proposals for extending Java with traits. For example, FeatherTrait Java (FTJ) [15] extends FJ [13] 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 [18], 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 [3] 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 [16]. 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 In certain situations, object interfaces allow automatic refinement for *return types*. This 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* [6], 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 [20] lights up this topic, proposing two new features: exact statements and nonheritable methods. Both are related to our work: 1) any method generated inside of the `of` method is indeed non-inheritable since there is no class name to extend from; 2) exact statements (a form of wild-card capture on the exact run-time type) could capture the "exact type" of an object even in a class-less environment. Admittedly, MyType greatly enhances the expressiveness and extensibility of object-oriented programming languages. Object interfaces use covariant-return types to simulate some uses of MyType. But our approach only works for refining return types, whereas MyType is more general, as it also works for parameter types. Nevertheless, as illustrated with our examples and case studies, object interfaces are still very useful in many practical applications.

Meta-Programming Competes with Language Extensions The most obvious solution to adding features to a language is via syntactic extensions. Syntactic extensions are often implemented as desugarings to the base language. For example, the Scala compiler was extended to directly support XML syntax. However, when syntactic extensions are independently created it is hard to combine multiple extensions into one. SugarJ [8] is a Java-based extensible language that aims at making the creation and composition of syntactic sugar extensions easy, by allowing programmers to extend Java with custom features (typically for DSLs). However SugarJ's goals are different from language tuning: SugarJ aims at creating and composing new syntax; whereas language tuning merely reinterprets existing syntax. It is clear that reinterpreting existing syntax only can be limiting for some applications. However, when language tuning is possible it has the advantage that existing tools for the language work out-of-the-box (since the syntax is still the same); and composition of independently developed tunings is straightforward.

Scala-Virtualized [19] is an extension to Scala, which allows blending shallow and deep embedding of DSLs. It redefines some of Scala's language constructs to method calls, which can be overridden by DSL implementer. Thus Scala-Virtualized can also reinterpret syntax, and be seen as a form of language tuning. However, although many Scala's language constructs are supported, not all language constructs can be virtualized.

When the base language has a flexible enough syntax and a fast and powerful enough reflection mechanism, we may just need to play with operator overloading and other language tricks to discover that the language feature we need can be expressed as a simple library in our language. An example of this is SQLAlchemy [2] in Python, which uses operator overloading to dynamically turn normal python expressions into database queries without requiring any syntactic extensions to Python. Java-like languages tend to sit in the middle of two extremes: libraries can not influence the type system, so many solutions valid in Python or other dynamic languages are not applicable, or have the cost of losing type-safety.

In Java-like languages compile time code generation comes at the rescue: if, for a certain feature (`@Obj` in our case), it is possible to use the original language syntax to *express/describe* any specific instantiation of such feature (annotating a class and providing getters), then we can insert in the compilation process a tool that examines and enriches the code before compilation. No need to modify the original source (for example we can work on temporary files). Java is a particular good candidate for this kind of manipulation since it already provide ways to define and integrate such tools in its own compilation process via annotation processing. In this way there is no need of temporary files, and there is a well-defined way of putting multiple extensions together.

Other languages offer even stronger support for safe code manipulation: Template

Haskell [24], F# (type providers) [3] and MetaFjig (Active Libraries) [23] all allow to execute code at compile time. They generate code that is transparently integrated in the program that is being generated/processed/compiled. In particular, MetaFjig offers a property called *meta-level-soundness*, ensuring by construction that library code (even if wrong or unreasonable) never generates ill-typed code. This is roughly equivalent to *Self coherence*, that we have to manually prove. Since MetaFjig is not working on annotated classes, there is not a "semantic with/without annotations". Our `@Obj` tuning theorem does not make sense in such context.

Formalization of Java 8 We provide a simple formalization for a subset of Java including default/static interface methods and object initialization literals (often called anonymous local inner classes). A similar formalization was drafted by Goetz and Field [12] to formalize defender (default) methods in Java. In their formalization, classes and interfaces can have only one method `m()` without arguments, so as to simplify method overloading and renaming. Classless Java is more general, as it supports multiple methods with arguments, it supports static methods, and features such as multiple inheritance of interfaces and reabstraction of default methods are also modeled.

9 Future work

In this section we discuss potential future work.

Qualifiers in Methods The biggest limitation of our approach is the inability to model qualifiers for class methods (private, protected, synchronized, etc.). For example, the absence of support for private/protected methods in Java 8 interfaces forces all members of interfaces to be public, including static methods. Since we use abstract methods to encode state, our state is always all public. Still, because the state can only be accessed by methods, it is impossible for the user to know if a certain method maps directly to a field or if it has a default implementation. If the user wants a constructor that does not directly maps to the fields, (as for secondary constructors in Scala) he can simply define its own `of` method and delegate on the generated one:

```
@Obj interface Point { int x(); int y();
   static Point of(int val){return Point.of(val,val);} }
```

However, the generated `of` method would also be present and public. If a future version of Java was to support *static private methods in interfaces* we could extend our code generation to handle also encapsulation. Currently, it is possible to use a public nested class with private static methods inside, but this is ugly and cumbersome. One possibility is that the annotation processor takes methods with a `@Private` annotation, and turns it into static private methods of a nested class. In this extension, also the `of` method could be made private following the same pattern.

Clone, toString, equals and hashCode Methods originally defined in Java class `Object`, as `clone` and `toString`, can be supported by our approach with special care. If an interface annotated with `@Obj` asks an implementation for `clone`, `toString`, `equals` or `hashCode` we can easily generate one from the fields.[4] However, if the user wishes to provide his own implementation, since the method is also implemented in `Object`, a conflict arises. The

[3] http://research.microsoft.com/fsharp/

[4] In particular, for clone we can do automatic return type refinement as we do for `with-` and fluent setters. Note how this would solve most of the Java ugliness related to `clone` methods.

generated code can resolve the conflict inside `of`, by implementing the method and delegating it to the user implementation, thus

```
@Obj interface Point { int x(); int y();
   default Point clone() { return Point.of(0,0); } } //user defined clone
```

would expand into

```
interface Point { ...//as before
   public static Point of(int _x, int _y) {
      return new Point() {...
         public Point clone() { return Point.super.clone();}};} }
```

10 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 define a subset of Java, called Classless Java, where programs and (reusable) libraries can be easily defined and used. 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. 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. Interestingly, without classes there is also no subclassing. This scratches an old itching point in the long struggle of subtyping versus subclassing: according to some authors, from a software engineering perspective, interfaces are just a kind of classes. Others consider more opportune to consider interfaces as pure types. We do not know how to conciliate those two viewpoints and Classless Java design. Classless Java does not have classes purely in the Java sense. Classless Java encourages coding in a more flexible way by either keeping a higher abstraction level (interfaces are a more abstract concept than classes), or relying on concrete object initialization (the `new I(){...}` construct).

More generally, we identify the concept of *language tuning*. We identify libraries that are already performing language tuning (Lombok and Cofoja), and we forecast many different kinds of language tuning will emerge on suitable platforms like Java or the C# CLR. We identify various kinds of safety guarantees that can be offered by language tuning, but the door is open for more flavors of safety guarantees to emerge.

References

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

2 Michael Bayer. Sqlalchemy - the database toolkit for python. `http://www.sqlalchemy.org`.

3 Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Sci. Comput. Program.*, 78(5):521–541, 2013.

4 Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *PPPJ'14*, 2014.

5 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.

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

7 Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.

8 Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *OOPSLA'11*, 2011.

9 Martin Fowler. Fluentinterface, December 2005. `http://martinfowler.com/bliki/FluentInterface.html`.

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

11 Brian Goetz. Allow default methods to override object's methods. Discussion on the lambda-dev mailing list, 2013. `http://mail.openjdk.java.net/pipermail/lambda-dev/2013-March/008435.html`.

12 Brian Goetz and Robert Field. Featherweight defenders: A formal model for virtual extension methods in java, 2012. `http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf`.

13 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

14 Nhat Minh Le. Contracts for java: A practical framework for contract programming. `https://cofoja.googlecode.com/files/cofoja-20110112.pdf`.

15 Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2):11, 2008.

16 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA '09*, 2009.

17 Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

18 John Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *International workshop on foundations and developments of object-oriented languages*, 2006.

19 Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-virtualized: Linguistic reuse for deep embeddings. *Higher Order Symbol. Comput.*, 25(1):165–207, 2012.

20 Chieri Saito and Atsushi Igarashi. Matching mytype to subtyping. *Sci. Comput. Program.*, 78(7):933–952, 2013.

21 Markku Sakkinen. Disciplined inheritance. In *ECOOP'89*, 1989.

22 Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03*, 2003.

23 Marco Servetto and Elena Zucca. Metafjig: a meta-circular composition language for java-like classes. In *OOPSLA'10*, 2010.

24 Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *ACM SIGPLAN Haskell Workshop*, 2002.

25 Philip Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.

26 Yanlin Wang and Bruno C. d. S. Oliveira. The expression problem, trivially! Technical report, HKU TR-2015-08, 2015. `http://www.cs.hku.hk/research/techreps/document/TR-2015-08.pdf`.

27 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL'05*, 2005.

28 Elena Zucca, Marco Servetto, and Giovanni Lagorio. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP'09*, 2009.

29 Reinier Zwitserloot and Roel Spilker. Project lombok. `http://projectlombok.org`.