Classless Java

John Q. Open[1] and Joan R. Access[2]

1 **Dummy University Computing Laboratory**
Address, Country
open@dummyuni.org

2 **Department of Informatics, Dummy College**
Address, Country
access@dummycollege.org

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 multiple inheritance quickly becomes cumbersome, and appears to be quite restricted.

This paper shows that, with some minor syntactic sugar, default methods and interface inheritance are in fact very expressive. Our proposed syntax sugar, called *object interfaces*, enables powerful object-oriented idioms, using multiple inheritance, to be expressed directly in Java. Object interfaces avoid some syntactic boilerplate caused by Java being originally designed to work mainly with classes. Moreover, object interfaces come with their own object instantiation mechanism, providing an alternative to class constructors. As a result many Java programs can be built without defining a single class! 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 and case studies. BRUNO: a first draft of the abstract. still missing "language tuning".

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Dummy keyword – please provide 1–5 keywords

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

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* [27]. The original notion of traits by Scharli et al. prescribes, among other things, that: 1) a trait provides a set of methods that implement behaviour; 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. [2]. 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"" [13]. 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.

© John Q. Open and Joan R. Access;
licensed under Creative Commons License CC-BY

Still, the design has been criticised for being too conservative and not very useful in its current form to model traits or 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 latter implements those methods.
- *Useful, general purpose methods, require special care in the presence of subtyping.* Methods such as *clone*, or *fluent* setters [10], not only require access to the internal state of an object, but they also require their types to be refined in subtypes.

Clearly, a way around those difficulties would be to try to change Java and just remove these limitations. Scala's own notion of traits, for example, allows state in traits. Of course adding state (and other features) to interfaces would complicate the language and require changes in 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 some minor syntactic sugar, default methods and interface inheritance are in fact be very expressive. Our proposed syntax sugar enables powerful object-oriented idioms, using multiple inheritance, to be expressed directly without modifications to the current Java language. We call our sugar *object interfaces*, because such interfaces can be instantiated directly, without the need for an explict class definition. Object interfaces come with their own object instantiation mechanism, providing an alternative to class constructors. Moreover, object interfaces avoid some syntactic boilerplate caused by Java being originally designed to work mainly with classes. This includes support for various common utility methods (such as getters and setters, or clone methods), even in the presence of subtyping. As a result, with object interfaces, many Java programs can be built without using a single class! BRUNO: integrate more of Marco's text.

To formalize object interfaces, we propose Classless Java (CJ)MARCO: I propose to call it directly ClassLess Java, the Featherweight part is in being classless.: a FeatherweightJava-style calculus, which captures the essence of interfaces with default methods. The semantics of object interfaces is given as a type-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-preserving, ensuring that the translation does not introduce type-errors. One interesting aspect of the theorem is that, if we ignore the object annotations, the program still type-checks under the regular typing rules of CJ (and Java).BRUNO: double-check! MARCO: I do not understand what you mean. 1) annotations are part of java and CJ, so you do not need to remove them 2) programs well typed without expansion are well typed with expansion, here you seams to state the opposite CJ's usefulness goes beyond serving as a calculus to formalize object interfaces. During the development process of CJ, we encountered an incoherent behaviour in Java's implementation of default methods that could be classified as a bug. BRUNO: Should we report bug to the Java developers mailing list? MARCO: we need to put the exact javac version We also present a prototype implementation of object interfaces using Java annotations. One benefit of an annotation-based implementation is that existing Java tools (such as IDE's) work with 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. MARCO: I removed your text here and I add after to try talking about tuning

To evaluate the usefulness of object interfaces, we illustrate 3 applications and case studies. The first application is a simple solution to the Expression Problem [?], supporting independent extensibility [34], and without boilerplate code. The second application is to show how embedded DSLs using fluent interfaces [] can be easily defined using object interfaces. Finally, 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 [2] using plain Java 8. Noteworthy, is the fact that all applications are implemented without defining a single class.

Moreover, we believe that our approach is an example of a more general idea, that we call *Language Tuning*, a concept that sits in the middle between a lightweight language extension and a glorified library. Thanks to Java dynamic class loading and annotation processing, annotations can trigger type-driven transformations on classes during either loading or compilation. There are many examples of such Language Tuning libraries, [?, lomboc,anotherImSearching] Language tuning can offers many features usually implemented by a real language extension, but they do not modify the syntax; pre-existing tools can work transparently on the tuned language.

In summary, the contributions of this paper are:

- **Object Interfaces:** A simple feature that allows various powerful programming idioms to be expressed conveniently in Java. Object interfaces are pure sugar: they are desugared into standard Java code, without loss of modularity or type-safety.
- **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 desugaring of object interfaces. We prove ...
- **A Bug in the Implementation of Java 8:**BRUNO: state this as a contrib?MARCO: yes, especially if we "suggest solutions"
- **Implementation and Case Studies:** We have a prototype implementation of object interfaces, using Java annotations. Moreover the usefulness of object interfaces is illustrated through various examples and case studies.
- **Language Tuning** We identify this concept and we describe Project Lomboc and object interfaces as examples of Language Tuning.

2 A Running Example: Animals

MARCO: fig1 is now better. Where we cite it in the text? can we make it so that is more "linear"? point2d <- point3D, animal <- (horse over bird) <- pegasus To propose a standard example, we show `Animals` with a two dimensional `Point2D` representing their `location`. Some kinds of animals are `Horses` and `Birds`. Birds can `fly`, thus their location need to be a three dimensional `Point3D`. Finally, we model `Pegasus` as a kind of `Animal` with the skills of both `Horses` and `Birds`.[1]

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

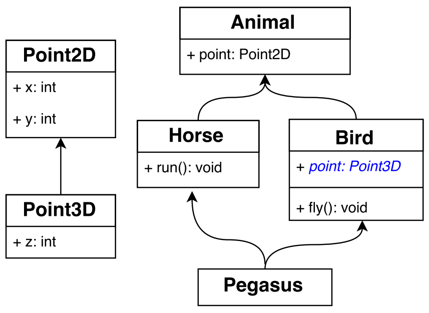

Figure 1 Pegasus Example.

2.1 `Point2D`: simple immutable data with fields

Since in ClassLess Java there are no classes, our `Point2D` must be an interface, as for example:

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

That is, since Java disallows fields inside interfaces, we simulate state by abstract methods. By using the new Java8 **`default`** methods, we can easily add implemented methods inside the interface. For example:

```
interface Point2D{ int x(); int y();
  default float distanceFromOrigin(){ return Math.sqrt(x()*x()+y()*y());}
  }
```

resemble a class that have fields (getters) and methods. However, how can we create an instance of `Point2D` without using classes? we can just provide an implementation for `x()` and `y()` as in

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

However we are not required to use this cumbersome syntax[2] for every object allocation. As programmes do, for ease or reuse, we can encapsulate boring long repetitive code in a method. A static factory method in this case is appropriate:

```
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;}};}
  }
```

This obvious constructor code can be automatically generated by our **@Mixin** annotation.

By annotating the interface `Point2D`, the annotation will generate a variation of the shown static method `of`, mimicking the functionality of a simple minded constructor: By looking the the methods that need implementation it first detects what are the fields, then generate an `of` method with one argument for each of them.

That is, we can just write

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

More precisely, a field or factory parameter is generated for every no-args method requiring implementation whose name does not have special meaning[3]. An example of code using

[2] Available in Java from version ...
[3] Formal definition of **@Mixin** behaviour is provided later.

Point2D is Point2D.of(42,myPoint.y()) where we return a new point, using 42 as x-coordinate, and taking all the other informations (only y in this case) from another point. This pattern is very common when programming with immutable data-structures; it is so common that we decided to support it in our code generation as with- methods, that is:

```
@Mixin interface Point2D{ int x(); int y();
  Point2D withX(int val); Point2D withY(int val);}
```

is expanded by our annotation into[4]

```
interface Point2D{ int x(); int y();
  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;}
    Point2D withX(int val){ return Point2D.of(val,this.y());}
    Point2D withY(int val){ return Point2D.of(this.x(),val);}
    };}
  Point2D withX(int val); Point2D withY(int val);}
```

If the programmer would like a different implementation, they can just provide it using **default**; that is:

```
@Mixin interface Point2D{ int x(); int y();
  default Point2D withX(int val){ ... }; default Point2D withY(int val){ ... }}
```

is expanded into

```
interface Point2D{ int x(); int y();
  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;}
    };}
  default Point2D withX(int val){ ... } default Point2D withY(int val){ ... }}
```

That is, we can provide implementation for methods in an interface, and our annotation generate code only if the method *needs* implementation. In this way is trivial for the programmer to personalize the behaviour if they have any special need.

2.2 Animal and Horse: simple mutable data with fields

Points2D are mathematical entities, thus we chosen to use immutable data structure to model them; however animals are real world entities, and when an animal moves, it is the same animal that have now a different location. We model this with mutable state.

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

MARCO: TODO: change point into "location" Here we declare abstract getter and setter for the mutable "field" location. As you see, we do not apply the **@Mixin** annotation. This is morally equivalent to an abstract class in full Java; that is we do not provide a convenient way to instantiate it.

[4] Note how we actually generate a real field **int** x=_x;. This provide a more uniform translation that can work also for mutable datastructures, where setters are required.

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

For `Horse`, concrete implementation of `run()` method need to be defined in a default method, where we also show the convenience of *with* methods.

2.3 `Bird`: field type refinement

`Point3D` and properties updater

`Bird` needs a 3D location, thus first of all we define `Point3D` and as follows:

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

Just to give you a feeling on how much boring code **@Mixin** is generating, we show the expanded code without then **@Mixin** annotation[5].

```
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());
        }    };  } }
```

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 and behave as expected. If the programmer wishes to suppress this behaviour and keep the signature as it was, it is sufficient to redefine the `with-` methods in the new class repeating the old signature. Again, the philosophy is that if the programmer provides something directly, **@Mixin** do not touch it. `with-` methods (functionally) update a field/property at a time. This can be inefficient, and sometime hard to maintain. Often we want to update many fields at the same time, 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 and in the current object update all field in that type that match fields in the current type. The idea is that we want as result something that is still like **`this`**, but modified to be as much as possible similar to the parameter. The cast in `with(Point2D)` is trivially safe since it is guarded by

[5] As you can see, the generated code is very repetitive, writing such code by hand can easily induce bugs, for example a distract programmer may swap the arguments of one of the many calls of `Point3D.of`.

an `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`**.

Unsatisfactory class based solutions to field type refinement

`Birds` are `Animals`, but while `Animals`, only need 2D locations, `Birds` need 3D locations. In Java if we 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 that is needed, and also it may requires the programmer to predict the future, or it may require to adapt 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 at-hoc, requires to basically duplicate the difference between `Point2D` and `Point3D` inside of `Bird`. Again, there are many reasons this would be bad, the most dramatic is that it would not scale to a scenario when the programmer of `Bird` and the programmer of `Point3D` are different.
- 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.

Many readers would now think that a language extension is needed. Instead, with our language *restriction* the problem can be easily challenged[6].

Our approach to field type refinement

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

In our proposed solution,

- by covariant method overriding we refine the field type
- by *overloading* we define a new setter with the more precise type
- by default method we provide an implementation for the setter called with the old signature.

From the type perspective, the key is the covariant method overriding; however from the semantic perspective the key is the implementation for the setter with the old signature: by using the `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 informations about what fields sits in `Point3D` and what in `Point2D` is properly encapsulated in the `with` method, and is transparent for the implementer of `Bird`.

[6] Often in programming languages "freedom is slavery".

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
$field$	$::=$	$T\ f\!=\!x;$	field declaration
$\mathcal{I}$	$::=$	$ann\ \texttt{interface}\ I_0\ \texttt{extends}\ \overline{I}\{\ \overline{meth}\ \}$	interface declaration
$meth$	$::=$	$\texttt{static}\ mh\{\ \texttt{return}\ e;\} \mid \texttt{default}\ mh\{\ \texttt{return}\ e;\} \mid mh;$	method declaration
mh	$::=$	$T_0\ m\ (T_1\ x_1 \ldots T_n\ x_n)$	method header
ann	$::=$	$\texttt{@Mixin} \mid \emptyset$	annotations
Γ	$::=$	$x_1{:}I_1 \ldots x_n{:}I_n$	environment

Figure 2 Grammar of ClassLess Java

2.4 `Pegasus` and multiple Inheritance

The *"multiple inheritance"* case appears at interface `Pegasus`. Pegasus (one of the best known creatures in Greek mythology) can not only *run* but also *fly*! It would be handy if they could obtain `fly` and `run` functionality from both `Horse` and `Bird`.

Using interfaces and **@Mixin** annotation, this can happen transparently.

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

Note how even the non trivial pattern for field type refinement is transparently composed, and our pegasus have a `Point3D location`. Note that this works because `Horse` do not perform any field type refinement , otherwise we may have to choose/create a common subtype in order for pegasus to exists.

3 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 calculus also supports a minimal extension to Java to account for the **@Mixin** annotation for object interfaces. The syntax, typing rules and method lookup mechanisms of CJ are modelled. Finally, using our formalization, we discuss examples of real Java programs that seem to be incoherentBRUNO: revise later.

3.1 Syntax and Typing

MARCO: Future work: updating multiple fields in one method call, `with(T v)`

Syntax Figure 2 shows the syntax of ClassLess Java. [7] The syntax formalizes a minimal version of Java 8, focusing on interfaces, default methods and object creation literals. There is no syntax for classes. 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 are super calls ($I.\texttt{super}.m(\overline{e})$), whose semantic is to call the (non static) method m over the **`this`** receiver, but statically dispatching to the version of the method defined in the interface I. A simple form of field updates ($x\!=\!e;e'$) is also modelled. In the

[7] To be compatible with java, the concrete syntax for an interface declaration with empty supertype list $I_1 \ldots I_n$ would also omit the **`extends`** keyword.

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. Note how our language is a subset of Java 8. The single non-Java 8 piece of syntax is the **@Mixin** annotation, which is the only one interesting piece of syntax in this article. Following standard practise, we consider a global Interface Table (IT) mapping from interface names I to interface declarations 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 defining 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 $\overline{meth}(m) = meth$.

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 $I <: I'$. From the interface table, we can read off the subtype relation between interfaces. The subtype relation is given by the **extends** clauses in the interfaces. We omit the definition of the usualBRUNO: where can it be found? add reference here?, traditional subtyping relation between interfaces.[8]BRUNO: is a footnote here the appropriate place to have this discussion? 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, respectivelly, extracting the method name and the return type from a method header. $\mathsf{mbody}(m, I)$, defined in Section **??**, 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 on 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 3 we show the typing rules. We discuss first 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. Secondly, we check that all method bodies are well-typed. To do this the enviroment 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. Finally, we need to check that all method headers are valid with respect to the superinterfaces of I; and we also need to check that all abstract and conflicted methods (that is methods, that need to be explicitly overriden) in the superinterfaces have been implemented. The definitions $\mathsf{sigvalid}$ and $\mathsf{alldefined}$ take care of this:

[8] Notice how there are no classes, thus there is no subclassing. We believe that this approach may scratch 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 are pure types. In this vision our language would have no subclassing. We do not know how to conciliate those two viewpoints and ClassLess Java design. We do not have Classes purely in the Java sense.

(T-Invk)
$$\frac{\begin{array}{c}\Gamma \vdash e \in I \\ \forall i \in 1..n \quad \Gamma \vdash e_i \in \_ <: I_i \\ \mathsf{mtype}(m, I) = \overline{I} \to I'\end{array}}{\Gamma \vdash e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \in I'}$$

(T-StaticInvk)
$$\frac{\begin{array}{c}\forall i \in 1..n \quad \Gamma \vdash e_i \in \_ <: I_i \\ \mathsf{mtypeS}(m, I) = \overline{I} \to I'\end{array}}{\Gamma \vdash I\texttt{.}m\texttt{(}\overline{e}\texttt{)} \in I'}$$

(T-SuperInvk)
$$\frac{\begin{array}{c}\Gamma(\texttt{this}) <: I \\ \forall i \in 1..n \quad \Gamma \vdash e_i \in \_ <: I_i \\ \mathsf{mtype}(m, I) = \overline{I} \to I'\end{array}}{\Gamma \vdash I\texttt{.super.}m\texttt{(}\overline{e}\texttt{)} \in I'}$$

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

(T-Obj)
$$\frac{\begin{array}{c}\forall i \in 1..k \quad \Gamma(x_i) <: T_i \\ \forall i \in 1..n \quad \Gamma, f_1{:}T_1, \ldots, f_k{:}T_k, \texttt{this}{:}I, \Gamma^{mh_i} \vdash e_i \in \_ <: I^{mh_i} \\ \mathsf{sigvalid}(mh_1 \ldots mh_n, I) \qquad \mathsf{alldefined}(mh_1 \ldots mh_n, I)\end{array}}{\Gamma \vdash \texttt{new } I\texttt{()\{ } T_1\ f_1\texttt{=}x_1\texttt{;} \ldots T_k\ f_k\texttt{=}x_k\texttt{;}\ mh_1\texttt{\{ return } e_1\texttt{;\}} \ldots mh_n\texttt{\{ return } e_n\texttt{;\}\}} \in I}$$

(T-update)
$$\frac{\begin{array}{c}\Gamma \vdash e \in \_ <: \Gamma(x) \\ \Gamma \vdash e' \in I\end{array}}{\Gamma \vdash x\texttt{=}e\texttt{;}\, e' \in I}$$

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

Figure 3 CJ Typing

$$\begin{array}{lcl}\mathsf{sigvalid}(mh_1 \ldots mh_n, I) & = & \forall i \in 1..n \quad mh_i <: \mathsf{mbody}(m^{mh_i}, I) \\ \mathsf{alldefined}(mh_1 \ldots mh_n, I) & = & \forall m \text{ such that } \mathsf{mbody}(m, I) = mh; \exists i \in 1..n\ m^{mh_i} = m\end{array}$$

BRUNO: Marco, you probably want to double-check the explanation. Any more explanation needed? BRUNO: Also, according to the text above, mbody returns the full method body, so in sigvalid, shouldn't we use mtype instead? The same for alldefined, should it be mtype? Still not very confident about alldefined.

The rule ((T-INTF)) checks that an interface I is correctly typed. First we check that the body of all the default and static methods are well typed. Then we check that $\mathsf{dom}(I)$ is the same of $\mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$. This is not a trivial check, since $\mathsf{dom}(I)$ is defined using mbody, that is 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 any method in both $\mathsf{dom}(I_i)$ and $\mathsf{dom}(I_j)$ $(i, j \in 1..n)$ is conflicting. MARCO: if we have space we can discuss the other rules too, but is not mandatory

3.2 Auxiliary Definitions

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

The body of a method m contains all the relevant information with respect to that method, like the type of m as well as the modifier. We use internally a special modifier conflicted to denote the case of two methods with conflicting implementation.

$\mathsf{mbody}(m, I_0) = \mathsf{override}(\overline{meth}(m), \mathsf{shadow}(m, \mathsf{tops}(\overline{I})))$
Where $IT(C_0) =$ ann `interface` I_0 `extends` $\overline{I}$`{` $\overline{meth}$`}`

As you can see, we are delegating the work to three others auxiliary functions:

$\mathsf{tops}(\bar{I}), \mathsf{shadow}(m, \bar{I})$ and $\mathsf{override}(meth, meth')$

MARCO: we need to discuss the names tops, shadow and override

tops recover from the interface table only the "needed" methods, that is, the non static ones that are not transitively reachable by following another, less specific, superinterface chain. Formally:

$meth \in \mathsf{tops}(m, \bar{I})$ iff $I \in \bar{I}$, $meth = \mathsf{mbody}(m, I)$, $meth$ not a static method
and $\nexists I' \in \bar{I} \setminus I$ such that $I' <: I$.

shadow choose the most specific version of a method, that is the unique version available, or a conflicted version from a set of possibilities. We do not model overloading, so it is an error if multiple versions are available with different parameter types. Formally:

$\mathsf{shadow}() = \mathsf{None}$
$\mathsf{shadow}(meth) = meth$
$\mathsf{shadow}(\overline{mh;}) = \mathsf{mostSpecific}(\overline{mh;})$
$\mathsf{shadow}(\overline{meth}) = \mathsf{conflicted}\, mh;$
where $\overline{meth}$ not of the form $\overline{mh;}$ and $\mathsf{mostSpecific}(\overline{meth}) \in \{mh;, \texttt{default } mh\texttt{\{return \_;\}}\}$
$\mathsf{mostSpecific}(\overline{meth}) = meth$
where $meth \in \overline{meth}$ and $\forall meth' \in \overline{meth} : meth <: meth'$,
$T\ m(\ T_1 x_1 \ldots T_n x_n) <: T' m(\ T_1 x_1' \ldots T_n x_n')$
where $T <: T'$
$meth <: \texttt{default } mh\texttt{\{return \_;\}} = meth <: mh;$
$\texttt{default } mh\texttt{\{return \_;\}} <: meth = mh; <: meth$

Where mostSpecific return the most specific method using return type specialization as introduced in JavaMARCO: insert the version number We just check the subtype between method headers, so we discard information abut method implementation.

The override function models how the implementation in an interface can override implementation in the superinterface; even in case of a conflict. Note how we use the special value None, and how (forth case) overriding can solve a conflict.

$\mathsf{override}(\mathsf{None}, \mathsf{None}) = \mathsf{None}$
$\mathsf{override}(meth, \mathsf{None}) = meth$
$\mathsf{override}(\mathsf{None}, meth) = meth$
where $meth$ not of the form $\mathsf{conflicted}\ mh;$
$\mathsf{override}(meth, meth') = meth$
where $meth' \in \{mh;, \texttt{default } mh\texttt{\{return \_;\}}, \mathsf{conflicted}\ mh;\}, meth <: meth'$

MARCO: The notation mtype get only non static methods, the notation mtypeS get only static ones

3.3 Method Overriding Mechanism

We summarize the mechanism for method overriding below (all methods considered here are methods with the same name and argument types):

1. Two abstract methods with subtyping relation do not cause conflict, and the most specific one wins. Subtyping relation for two methods means they have the same arguments' types and one return type is a subtype of the other one. For example:

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

Method `Integer m()` in `B` wins over method `Object m()` in `A` because `Integer <: Object`.

2. A default method conflict with another default method or abstract method. For example:

```
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 in contrast with what happens in most trait models, where `D2` would be accepted, and the implementation in `A2` would be part of the behaviour of `D2`.

3. The method in the current interface wins over any methods defined in its super-interfaces, provided that the method conform to the subtype of all methods in its super-interfaces, i.e., the method is the most specific one. For example:

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

3.4 Method Overriding Incoherence in Java

While trying to formally encode the Java specification we have done some tests to clarify corner case behaviour. Consider 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 we define a new interface `M` implementing one $\mathtt{A}_i$ and one $\mathtt{B}_i$? we have 9 cases, that can fit nicely a table:

M extends	**A1**	**A2**	**A3**
B1	conservative error	conflict error	conservative error
B2	both abstract, accepted	conservative error	both abstract, accepted
B3	accepted?	conflict error	conservative error

We try to classify the results out of the table:

- **conflict error** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ are both implemented. This is also considered an error in most trait models.
- **both abstract, accepted** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ are both abstract. This is also considered correct in all trait models.
- **conservative error** happens when the method from $\mathtt{A}_i$ and one $\mathtt{B}_i$ is implemented in only one side. The is different from what we would expect in a trait model, but it may make sense with the conservative idea that a method defined in an interface should not silently satisfy a method in another one.

We are very confused on the result for **B3**,**A1**, and we can not find any rational for this behaviour. [9]

[9] We refer to the behaviour of `javac` as in JDK version isMARCO: add.

To be coherent with the idea of **conservative error**, the case should not be accepted. We do not see how this should behave differently from **B1**,**A1**, and **B3**,**A3**. We fear that the only retro-compatible fix for this strange behaviour is to accept all the cases of **conservative error** in a future version of Java.

In our approach, we choose to not model this strange behaviour (a bug?). Our auxiliary function $\mathsf{mbody}(m, I)$ enforce the **conservative error** strategy. The rest of our formalization is parametric with the definition of $\mathsf{mbody}(m, I)$, thus if Java change its resolution strategy to a more permissive one, only minor adaptations in $\mathsf{mbody}(m, I)$ would be needed.

4 What @Mixin Generates

We now show what the **@Mixin** annotation generates. We present a formal definition for most of the generated methods; however in our formalism we do not consider casts or `instanceof`, so we do not include the `with` method. For the same reason we do not include `void` returning setters, 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.

4.1 Translation Function

$$[\![\texttt{@Mixin}\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \}]\!] = \emptyset\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \overline{meth}'\}$$
$$\text{where } \mathsf{valid}(I_0), \texttt{of} \notin \mathsf{dom}(\overline{meth}) \text{ and } \overline{meth}' = \mathsf{ofMethod}(I_0)\ \mathsf{otherMethods}(I_0, \overline{meth}).$$

To translate an annotated interface, we add the `of` method, and then we add some other methods. However, first of all we check if the interface is valid for annotation:

$\mathsf{valid}(I_0)$ holds if $\forall m \in \mathsf{dom}(I_0)$, if $mh; = \mathsf{mbody}(m, I_0)$, one case is satisfied: $\mathsf{isField}(meth)$, $\mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$

That is, we can categorize all the *not implemented* methods in a pattern that we know how to implement.

Moreover, we check that the method `of` is not already defined by the user. In our simplified formalization we consider this to be just an error. In our prototype we keep `overloading` into account, and so we check that an of method with the same signature of the one we would like to generate is not already present.MARCO: Do we check it?

In the following we will write `with#`m to append m to `with`, following the camelCase rule, so the first letter of m must be lower-case and is turned in upper-case upon merging. For example `with#foo`=`withFoo`. Special names $\mathsf{special}(m)$ are `with` and all the identifiers of form `with#`m.

4.2 ofMethod

We now formally define $\mathsf{ofMethod}$, the function that generates the method `of`, that behaves like a factory. To avoid boring digressions about well known ways to find unique names, for the sake of this formalization we assume that no-args methods do not start with underscore, and we prefix method names with underscore to obtain valid parameter names.

$$
\begin{aligned}
\mathsf{ofMethod}(I_0) = \;& \texttt{static}\ I_0\ \texttt{of(}I_1\ \_m_1\texttt{,}\ldots I_n\ \_m_n\texttt{)\{return new}\ I_0\texttt{()\{}\\
&\quad I_1\ m_1 = \_m_1\texttt{;}\ldots I_n\ m_n = \_m_n\texttt{;}\\
&\quad I_1\ m_1\texttt{()\{return}\ m_1\texttt{;\}}\ \ldots I_n\ m_n\texttt{()\{return}\ m_n\texttt{;\}}\\
&\quad \mathsf{withMethod}(I_1, m_1, I_0, \overline{e}_1)\ldots\mathsf{withMethod}(I_n, m_n, I_0, \overline{e}_n)\\
&\quad \mathsf{setterMethod}(I_1, m_1, I_0)\ldots\mathsf{setterMethod}(I_n, m_n, I_0)\\
&\quad \mathsf{withMethod}(I_0)\\
&\texttt{\};\}}
\end{aligned}
$$

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

HAOYUAN: Should we include the $\mathsf{withMethod}(I_0)$ in the formalization?

The function $\mathsf{fields}(I_0)$ (formally defined later) denotes all the fields in the current interface. For methods inside the interface with the form $I_i\ m_i\texttt{();}$

- m_i is the field name, and have type I_i.
- $m_i\texttt{()}$ is the getter, that just return the current field value.
- if a method $\texttt{with\#}m_i$ 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 parameter. This correspond to the expressions $\overline{e}_i$.
- $\_m_i\texttt{(}I_i\ \texttt{\_val)}$ is the setter. In our prototype we use name m_i, here we use the underscore to avoid modelling overloading.

4.3 Other auxiliary functions

$$
\begin{aligned}
\mathsf{withMethod}: \quad & \mathsf{withMethod}(I, m, I_0, \overline{e}) = I_0\ \texttt{with\#}m\texttt{(}I\ \texttt{\_val)\{return}I_0\texttt{.of(}\overline{e}\texttt{);\}}\\
& \text{iff } \mathsf{mbody}(\texttt{with\#}m, I_0) \text{ is of form } mh;\\
& \mathsf{withMethod}(I, m, I_0, \overline{e}) = \emptyset \text{ otherwise}\\
\mathsf{setterMethod}: \quad & \mathsf{setterMethod}(I, m, I_0) = I_0\ \_m\texttt{(}I\ \texttt{\_val)\{}\ m\texttt{= \_val;return this;\}}\\
& \text{iff } \mathsf{mbody}(\_m, I_0) \text{ is of form } mh;\\
& \mathsf{setterMethod}(I, m, I_0) = \emptyset \text{ otherwise}
\end{aligned}
$$

HAOYUAN: method name of setter? m or $\_m$?

As you can see above, `with-` and setter methods are generated if needed. We can discover if there is the need of generating such methods by checking if the method is unimplemented in I_0. Note that we do not need to check if its header is a subtype of what we would generate, this is ensured by $\mathsf{valid}(I_0)$.

$$
\begin{aligned}
\mathsf{otherMethods}: \quad & I_0\ \texttt{with\#}m\texttt{(}I\ \texttt{\_val);} \in \mathsf{otherMethods}(I_0, \overline{meth})\\
& \text{iff } I\ m\texttt{();} \in \mathsf{fields}(I_0), \mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m, I_0))\\
& \ \text{and } \texttt{with\#}m \notin \mathsf{dom}(\overline{meth})\\
& I_0\ \_m\texttt{(}I\ \texttt{\_val);} \in \mathsf{otherMethods}(I_0, \overline{meth})\\
& \text{iff } I\ m\texttt{();} \in \mathsf{fields}(I_0), \mathsf{isSetter}(\mathsf{mbody}(\_m, I_0))\\
& \ \text{and } \_m \notin \mathsf{dom}(\overline{meth})
\end{aligned}
$$

Other methods that we need to generate in the interface are `with-` and setters. This is needed only if we need to refine the return type. To discover if this is the case, we check if such `with-` or setter is required by I_0, but is not already present in the methods directly declared in I_0.

fields :	$meth \in \mathsf{fields}(I_0)$ iff $meth \in \mathsf{dom}(I_0)$ and $\mathsf{isField}(meth)$
isField :	$\mathsf{isField}(I\ m\texttt{();})$ if not $\mathsf{special}(m)$
isWith :	$\mathsf{isWith}(I'\ \texttt{with\#}m\texttt{(}I\ x\texttt{);}, I_0)$ if $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m\texttt{();}$ and not $\mathsf{special}(m)$
isSetter :	$\mathsf{isSetter}(I'\ \_m\texttt{(}I\ x\texttt{);}, I_0)$ if $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m\texttt{();}$ and not $\mathsf{special}(m)$

We have not formally modelled non fluent setters and the `with` method; informally

- For methods inside the interface with the form `void` $m\texttt{(}I\ x\texttt{);}$:
  - Check if exist method $I\ m\texttt{();}$. If not, generate error (that is, is not $\mathsf{valid}(I_0)$).
  - Generate implemented setter method inside `of`:
    `public` `void` $m\texttt{(}I\ \texttt{\_val)\{}\ m\texttt{=\_val;\}}$ Note how 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(`$I\ x$`);`:
  - As for 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($e_1 \ldots e_n$);}
    ```
    where with $m_1 \ldots m_n$ fields of I_0, e_i =`_val.`m_i`()` if I has a m_i`()` method; otherwise $e_i = m_i$.
  - If needed, as for `with-` and setters, generate the method header with refined return type in the interface.

MARCO: insert somewhere description of fluent setter [?, ?]. This allows for convenient and chains of setters, as we will show later MARCO: insert forward reference when available.

4.4 Results

THEOREM. For a given $\mathcal{I}_0 \ldots \mathcal{I}_n$ interface table such that $\forall \mathcal{I} \in \mathcal{I}_0 \ldots \mathcal{I}_n, \mathcal{I}$ OK, then in the interface table $[\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ $\forall \mathcal{I} \in [\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ either $\mathcal{I}$ OK or $\mathcal{I}$ is a subtype of $\mathcal{I}_0$.

HAOYUAN: In theorem: I_0 is valid and $\texttt{of} \notin \mathsf{dom}I_0$. To prove the theorem we need an extra lemma: if the translation is well defined, then $[\![\mathcal{I}_0]\!]$ is well formed.

To understand this theorem statement, we need to understand that there are three kind of guarantees that we can offer for safety:

- *Self coherence*: the generated code itself is well-formed; there are no type errors.
- *Client coherence*: the client code doesn't fail if it was not failing before code generation.
- *Heir coherence*: the heirs are safe.

The first two levels of safety are essential to our lightweight framework, nevertheless, the third one is not needed. In Java it implies not touching the annotated interface, and surely an alternative approach is to create a new interface and generate what we need. But since we are using Lombok for the translation, the transformation is applied to the annotated interface itself. But heir coherence is indeed an issue. See below how this is affected by our approach:

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

Here is the client code for our example. With interface `B` annotated with **@Mixin** , by the translation rule, it will generate a method "`B withX(int x);`" inside, as an automatic return type refinement. But in that case, interface `C` doesn't type check.

Now we go back and read the statement of our theorem again. It illustrates that a program with the translation of **@Mixin** applied still type checks if it did, except for the subtypes of the annotated interface (type safety of them is not guaranteed).

HAOYUAN: Text needs improving.

To prove the theorem we introduce two lemmas below. The complete proof is available in Appendix A.1, A.2 and A.3.

LEMMA 1. Assume that

$$\begin{aligned} \mathcal{I}_0 &= \texttt{@Mixin}\ \texttt{\textbf{interface}}\ I_0\ \texttt{\textbf{extends}}\ \overline{I}\,\{\ \overline{meth}\ \} \\ [\![\mathcal{I}_0]\!] &= \emptyset\ \texttt{\textbf{interface}}\ I_0\ \texttt{\textbf{extends}}\ \overline{I}\,\{\ \overline{meth}\ \overline{meth}'\} \end{aligned}$$

If $\mathcal{I}_0$ satisfies $\mathsf{dom}(I_0) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$, then $[\![\mathcal{I}_0]\!]$ satisfies $\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth}) \cup \mathsf{dom}(\overline{meth}')$.

LEMMA 2. If $\mathcal{I}_0$ OK, then $[\![\mathcal{I}_0]\!]$ OK.

5 Implementation

One of the downside of Java is the code verbosity, e.g., getter and setter methods for fields, unneeded type annotation, etc. Lombok project [] is a Java tool that aims at removing (or reducing) these boilerplate code as far as possible, 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. By realizing these transformations, with new annotations supported, users can generate new field and method members, and inject them to current class types automatically at compile time. More specifically, a transformation is based on a handler, which acts on the AST from parsing 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.

Our implementation is based on extension to Lombok. In Eclipse, with an interface annotated by **@Mixin** , the automatic annotation processing is performed and the information of the interface from compilation is captured in the "Outline" window, including 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 content assist autocomplete will work for the newly generated methods.

The biggest reasons to use Lombok rather than using conventional Java annotation processor is

- Lombock modify the generation process of the class files. 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.This is the ability that conventional Java annotation processor do not provide.

Our prototype implementation using Lombok has certain limitation:

- We do not support separate compilation yet at this stage, for in the implementation it is hard to capture a type declaration from its reference.
- At this stage our implementation only realizes the Eclipse handler and our experiments are all conducted in Eclipse.

6 Case Studies

Haoyuan and Yanlin

6.1 A Trivial Solution to the Expression Problem

The *Expression Problem* (EP) [31] is coined by Wadler about modular extensibility issues in software evolution and has been a hot topic in programming languages since. Today we know of various solutions to the EP that either rely on *new programming language features* [6, 7, 18, 21, 5, 20, 12, 33, 17, 32], or can be used as *design patterns* [11] in existing languages [30, 24, 28, 22, 23]. In this paper we show that the EP can be solved with our **@Mixin** annotation in a trivial way. We will use a canonical example of implementing arithmetic expressions gradually to illustrate the usage.

6.1.1 Initial System

The initial system expresses arithmetic expressions with only literals and addition structure:

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

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

6.1.2 Adding a New Variant

It is easy to add new data variants to the code in the initial system in a modular way. For example, the following code shows how to add the subtraction variant.

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

6.1.3 Adding a New Operation

Adding new operations to the previous system is still straightforward (although requires longer code). The following code shows an example of adding a new operation `print`.

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

The basic idea is to define interfaces for extending old interfaces. The interface `ExpP` extending interface `Exp` is defined with the extra function `print()`. Interfaces `LitP` and `AddP` are defined with default implementations of operation `print()`, extending base interfaces `Lit` and `Add`, respectively. Importantly, note that in `AddP`, the types of "*fields*" `e1,e2` are refined via the *return types refinement* of getter methods `e1()` and `e2()`.

6.2 A Small Game Example

We now present a variant of a classic game example, which is often used to evaluate reuse ability related to inheritance and design patterns. The code using our approach and the solution by Bono et. al [2] in online, for space constraints, we will omit it here. 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 behaviour), multiple inheritance (a special kind of door may require features from two other door types). It also shows how to model symmetric sum, override and alias as trait-oriented programming. Moreover, the code is provided in [2] so that we can have a clear and fair comparison with our approach, as shown in Table 1.

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

Table 1 Code Reduction and Reuse.

6.3 Other case studies

MARCO: should we put here the DB with fluent setters?

7 Related Work

In this section we discuss related work and comparison to Classless Java.

7.1 Multiple Inheritance in Object oriented languages

Many authors have argued in favour or against multiple inheritance. It provides expressive power, but it is difficult to model, and implement and can create programs that are hard to reason about. These difficulties include the famous diamond (fork-join) problem [3, 26], conflicting methods, and the yo-yo problem [29]. To conciliate the need for expressive power and the need for simplicity, lots of models have been proposed in the past few years, including mixins [3] and traits [27]. They provide novel programming architecture models in the OO paradigm.

1. Mixins allows to name components that can be applied to various classes as reusable functionality units. However, they suffer from linearisation: the order of mixin application is relevant in often subtitle and undesired ways. constraints. This hinders their usability and their ability of resolving conflicts: the linearisation (total ordering) of mixin inheritance cannot provide a satisfactory resolution in all cases and restricts the flexibility of mixin composition.

   To fight those limitations, an algebra of mixin operators is introduced [1], but this raised the complexity of the approach, especially when constructors and fields are considered [35].
2. Simplifying the mixin algebras approach, traits draw a strong line between units of reuse (traits) and object factories (classes) In this model, traits [27] units of reusable code, containing only methods as reusable functionalities. Thus, no state/state initialization is considered.

   Classes act as object factories, requiring functionalities from multiple traits. Traits offers a trait algebra with operations like sum, alias and exclusion, provided for explicit conflict resolution.

   Concluding, (pure) traits do not allow state and they do not offers any reuse instrument to ensure that state is coherently initialized when finally defined in classes. Traits can't be instantiated. This model requires two concepts (traits and classes) to coexist and cooperate.

   Some authors 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.
3. C++ and Scala also try to provide solutions to multiple inheritance, but both suffer from object initialization problems. Virtual inheritance in C++ provides another solution to multiple inheritance (especially the diamond problem by keeping only one copy of the base class) [8], however suffers from object initialization problem as pointed out by Malayeri et al. [19]. It bypasses all constructor calls to virtual superclasses, which would potentially cause serious semantic error. Scala solution (very similar to linearised mixins, but misleadingly called traits in the language) avoids this problem by disallowing constructor parameters, causing no ambiguity in cases such as diamond problem. This approach has limited expressiveness, and suffers from all the problems of linearised mixin composition. Python also offers multiple inheritance via linearised mixins. Indeed python any class is implicitly a mixin, and mixin composition informally expressed as

   ```
   class A use B,C {...new methods...}
   ```

   can be expressed in python as

   ```
   class Aux: ...new methods...
   class A(B,C,Aux): pass
   ```

7.2 Multiple Inheritance in Java

Since Java 8 default methods are introduced, concrete method implementation are allowed to be defined (via the **default** keyword) inside interfaces. Since Java supports implementation of multiple interfaces (instead of extension of a single class), the introduction of default methods opens the gate for various flavours of multiple inheritance in Java, using interfaces. Former work by Bono.et. al. [2]. provides details on mimicking traits through interfaces.

There are proposals for extending Java (before Java8) with traits. For example, Feather-Trait Java (FTJ) [16] by Liquori et al. extends the calculus of Featherweight Java (FJ) [15] with statically-typed traits, adding trait-based inheritance in Java. Except for few, mostly syntactic details, their work can be emulated/rephrased with Java8 interface.

7.3 Multiple Inheritance in ClassLess Java

Our approach is based on forbidding the use of certain language constructs (classes), in order to rely on the modular composition offered by the rest of the language (interfaces). Since Java was designed for classes, a direct class-less programming style is verbose and feel 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 object oriented feel. We consider our approach as an alternative to traits or mixins.

A comparison on how to to emulate the original operations on traits using Java8 can be find in[2]. We briefly recall the main points of their encoding; however we propose a different representation of **exclusion**. The author of [2] agree that our revised version 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(); }
  ```

Besides, we support more features than the original trait model:

- We provide `of` methods for the annotated interfaces. During annotation processing time, the “fields” inside an interface are observed and a static method `of` is automatically injected to the interface with its arguments correspondingly. Such a method is a replacement to the constructors in original traits, making instantiation more convenient to use. That is, in our approach there are only interfaces, our model requires a single concept, while the trait model requires traits *and* classes.

- We provide `with-` methods as auxiliary constructors. A `with-` method is generated for each field, just like record update, returning the new object with that field updated Furthermore, we do automatic return type refinement for these kind of methods. This feature is comparatively useful in big examples, making operations and behaviours more flexible.
- We provide two options for generating setters. There are two kind of setters which are commonly used, namely *void setters* and *fluent setters*. The only difference is that a fluent setter returns the object itself after setting, thus supporting a pipeline of such operations. The generation depends on which type of setter is declared in the interface by users.

These are the additional features supported by our model, conversely, there are certain operations we cannot model, such as method renaming (as in [Reppy2006]), which breaks structural subtyping.

There are other limitations of our current approach, but they may be addressed in future work. See MARCO: reference to future work when there

7.4 ThisType/MyType/Extensibility

In certain situations, our approach allows 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* [4], 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 huge limitation in class based object oriented programming, and is exasperated by interface-only programming as we propose: no type is ever exact since classes are not part of the language. A recent article [25] lights up this topic, since they propose two features: exact statements YANLIN: do we support exact statements?MARCO: no we do not. When is that it may seams we do? and non-inheritable methods; both related to our work: any method generated inside of the `of` method is indeed non-inheritable, since there is no class name to extends from, and 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.

7.5 Meta-programming competes with Language extensions

The most obvious solution to add features to a language is language extension. They are often implemented as syntactic extensions that can be desugared to the base language. For example, the Scala compiler was extended to directly supports XML syntax. However, this approach does not support combining multiple extensions into one. We are de facto creating a fork in the language, and rarely the new fork gain enough traction to become the main language release. On this topic we mention SugarJ [9]; a Java-based extensible language allowing programmers to extend it with custom features by definitions in meta-DSLs (SDF, Stratego, etc).

On the other side, when the starting language have a flexible enough syntax and a fast and powerful enough reflection, 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. For example, consider SQL alchemy in python.

Java-like language tends to sits in the middle of this two extremes: libraries can not influence the type system, so many solutions valid in python or other languages could not be applicable, or may be applicable at the cost of loosing safety.

Here (compile/load time) code generation come at the rescue: if for a certain feature (**@Mixin** 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 provide getters), then we can insert in the compilation process a tool that exam and enrich 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: in this way there is no need of temporary files, and there is a well defined way of putting multiple extensions together.

Other languages offers even stronger support to safe code manipulation: Template Haskell [], F# (type providers) [] and MetaFjig (Active Libraries) [] all allows to execute code at compile time and to generate code/classes that are transparently integrated in the program that is being generated/processed/compiled. In particular, MetaFjig offers a property called *meta-level-soundness*. In short this property ensures by construction that library code (even if wrong or non nonsensical) would never generate ill-typed code. This is roughly equivalent to what we state and manually proof in Lemma 2 for our particular transformation. Since MetaFjig is not working on annotated classes, there is no direct equivalence on the overall theorem of safety we shown.

MARCO: text under still to revise. Sould all the lombock discussion go in the implementation section? – just to keep stuff together? Annotations/Code Generation/Lombok

What is the difference between our work and existing annotations? What is it that we can do that, those annotations cannot?

-Lombok [] project provides a set of predefined annotations, including constructor generators similar as ours (**@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 they are 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 **@Mixin** . However, none of these annotations can model what we are doing with **@Mixin** - generating a constructor for *interfaces*. Apart from constructors, **@Mixin** also provides other convenient features (including generating fluent setters, type refinement,etc) that the base Lombok project could not provide.

7.6 Fluent Interfaces

Talk about existing libraries that use fluent interfaces (do any of those use any code generation to assist in the implementation?).

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. For most languages, to create fluent interfaces, programmers have to either handwritten everything or create a wrapper around the original non-fluent interfaces, using **this**. In Java, several libraries help create fluent APIs easily (by code generation), including jOOQ, op4j, fluflu, JaQue, etc. These libraries have given us hints on the implementation of fluent setters in ClassLess Java.

7.7 Formalization of Java8

We provide a simple and well designed 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 byMARCO: I use the term draft because I seams to remember it was just a technical report, I'm right? Goetz and Field [14] to formalize defender (default) methods in Java. However this formalization is limited to model exactly one method inside classes/interfaces.

As a evidence of the attention and care present in our formalization work, while double-checking the behaviour of Java in side cases we have discovered a likely bug in the current `javac`.MARCO: refer to before when we explain the issue.

8 Future work

8.1 Private state

The biggest limitation of our approach is the absence in Java8 of support for private/protected methods in interfaces.That is, in Java8 all members of interfaces are public, including static methods. Since we use abstract methods to encode the state, our state is always all public; however is impossible for the user to know if a certain method maps directly to a field or if it have a default implementation. If the use 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, as in

```
@Mixin 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, is possible to use a public nested class with private static methods inside, but this is ugly and cumbersome. We are considering if our annotation processor can take code with `@Private` annotation and turn it into static private methods of a nested class. In this extension, also the of method could be made private following the same pattern.

8.2 State initialization

As discuss before, the user can trivially define its own `of` method, and initialize a portion of the state with default values. However, the initialization code would not be reused/reusable, and subinterfaces would have to repeat such initialization code. If a field has no setters, a simple alternative is to just define the “field” as a default method as in

```
@Mixin interface Box{ default int val(){return 0;} }
```

if setters are required, a possible extension of our code expansion could recognize a field if the getter is provided and the setter is required, and could generate the following code:

```
interface Box{
  default int val(){return 0;} //provided
  void val(int _val);//provided
  static Box of(){return new Box(){//generated
    int val=Box.super.val();
  int val(){return val;}
  void val(int _val){val=_val;}
  };}}
```

We are unsure of the value of this solution: is very tricky, the user define a method that (contrary to our usual expectation) is actually overridden in a way that the behaviour change, but change only after the first setter is called, plus this code would cache the result instead of re-computing it every time. This can be very relevant and tricky in a non functional setting.

8.3 Class invariants in ClassLess Java

Since the objects are created by automatically generated methods, another limitation of our current approach is that there is no place where the user can dynamically check for class invariants. In Java often we see code like

```
class Point{ int x; int y;
  Point(int x; int y){this.x=x;this.y=y; assert this.checkInvariant();}
  private boolean checkInvariant(){... x>0,y>0...}
```

We are considering an extension of our annotation where default methods with the special name `checkInvariant()` will be called inside of the `of` methods. if multiple interfaces are implemented, and more then one offers `checkInvariant()`, a composed implementation could be automatically generated, composing by `&&` the various competing implementations.

8.4 Clone, toString, equals and hashCode

Methods originally defined in `Object`, as `clone` and `toString` can be supported by our approach, but they need special care. If an interface annotated with **@Mixin** ask an implementation for `clone`, `toString`, `equals` or `hashCode` we can easily generate one from the fields.[10]

However, if the user wish to provide its own implementation, since the method is also implemented in `Object` we would have a conflict, that we have to explicitly resolve inside of `of`, by implementing the method and delegating it to the user implementation, thus

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

Would expand into

```
interface Point{ int x(); int y();
  default Point clone(){ return Point.of(0,0);}//user defined clone
  public static Point of(int _x,int_y){
    return new Point(){...
      public Point clone(){ return Point.super.clone();}
      }; } }
```

9 Conclusion

Before Java 8, concrete methods and static methods where not allowed to appear in interfaces. Java 8 allows static interface methods and introduces *default methods*, which allow for

[10] 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.

implementation insides interfaces. This had an important positive consequence that was probably overlooked by the Java design team: the concept of class (in java) is now 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 for some syntactic boilerplate caused by Java not being originally designed to work without classes, we introduce a new annotation:**@Mixin** provide default implementations for various methods (e.g. getters, setters, with-methods) and a mechanism to instantiate objects. **@Mixin** 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 **@Mixin** annotation can be less cumbersome than full Java.BRUNO: May need rewriting

References

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

2 Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ'14, 2014.

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

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

5 Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *ECOOP'98*, 1998.

6 Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17:805–843, November 1995.

7 Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00*, 2000.

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

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

10 Martin Fowler. Fluentinterface at http://martinfowler.com/bliki/fluentinterface.html, December 2005.

11 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

12 Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

13 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`.

14 Brian Goetz and Robert Field. Featherweight defenders: A formal model for virtual extension methods in java, 2012.

15 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), May 2001.

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

17 Andres Löh and Ralf Hinze. Open data types and open functions. In *PPDP '06*, 2006.

18 O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*, 1989.

19 Donna Malayeri and Jonathan Aldrich. *CZ: multiple inheritance without diamonds*, volume 44. ACM, 2009.
20 Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: new-age components for old-fashioned java. In *OOPSLA '01*, 2001.
21 Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06*, 2006.
22 Bruno C. d. S. Oliveira. Modular visitor components. In *ECOOP'09*, 2009.
23 Bruno C. d. S. Oliveira and William R. Cook. Extensibility for the masses: Practical extensibility with object algebras. In *ECOOP'12*, 2012.
24 Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. Extensible and modular generics for the masses. In *Trends in Functional Programming*, 2006.
25 Chieri Saito and Atsushi Igarashi. Matching mytype to subtyping. *Science of Computer Programming*, 2013.
26 Markku Sakkinen. Disciplined inheritance. In *ECOOP'89*, 1989.
27 Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP'03*, 2003.
28 Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423 – 436, 2008.
29 David H Taenzer, Murthy Ganti, and Sunil Podar. Problems in object-oriented software reuse. In *ECOOP*, volume 89, pages 25–38, 1989.
30 Mads Torgersen. The expression problem revisited – four new solutions using generics. In *ECOOP'04*, 2004.
31 Philip Wadler. The Expression Problem. Email, November 1998. Discussion on the Java Genericity mailing list.
32 Stefan Wehr and Peter Thiemann. JavaGI: The interaction of type classes with interfaces and inheritance. *ACM Trans. Program. Lang. Syst.*, 33, July 2011.
33 Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *ICFP '01*, 2001.
34 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL'05*, 2005.
35 Elena Zucca, Marco Servetto, and Giovanni Lagorio. Featherweight Jigsaw - A minimal core calculus for modular composition of classes. In *ECOOP'09*, number 5653, 2009.

A Appendix

A.1 Proof of LEMMA 1

LEMMA 1. Assume that

$$\begin{aligned}\mathcal{I}_0 &= \texttt{@Mixin}\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \} \\ [\![\mathcal{I}_0]\!] &= \emptyset\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \overline{meth}'\}\end{aligned}$$

If $\mathcal{I}_0$ satisfies $\mathsf{dom}(I_0) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})$, then $[\![\mathcal{I}_0]\!]$ satisfies $\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth}) \cup \mathsf{dom}(\overline{meth}')$.

Proof. It suffices to prove $\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_0) \cup \mathsf{dom}(\overline{meth}')$.

- If $m \notin \mathsf{dom}(\overline{meth}')$, by the definition of mbody, it is obtained from override. The first argument of override, namely $\overline{meth} \cup \overline{meth}'(m)$, is equal to $\overline{meth}(m)$, and the second argument is not changed as well, thus the result of override is not changed during translation. Hence $m \in \mathsf{dom}([\![\mathcal{I}_0]\!])$ iff $m \in \mathsf{dom}(I_0)$.

- If $m \in \mathsf{dom}(\overline{meth}')$, we are to prove $m \in \mathsf{dom}(\llbracket \mathcal{I}_0 \rrbracket)$. So if m is the `of` method generated by ofMethod, its mbody value is well defined, by the rule $\mathsf{override}(meth, \mathsf{None}) = meth$. On the other hand, if m is generated by otherMethods, namely m is a `with-` or setter method, we can also get its mbody value, since in the definition of otherMethods, we can see isWith and isSetter ensure the compatible subtyping relationship. Hence $m \in \mathsf{dom}(\llbracket \mathcal{I}_0 \rrbracket)$ as well.

◀

A.2 Proof of LEMMA 2

LEMMA 2. If $\mathcal{I}_0$ OK, then $\llbracket \mathcal{I}_0 \rrbracket$ OK.

Proof. By the rule (T-INTF) in Figure 3, we divide it into two parts.
Part I. For each default or static method in the domain of $\llbracket \mathcal{I}_0 \rrbracket$, the type of the return value is compatible with the method's return type.

Since $\mathcal{I}_0$ OK, all the existing default and static methods are well typed in $\llbracket \mathcal{I}_0 \rrbracket$, except for the new method `of`. It suffices to prove that it still holds for $\mathsf{ofMethod}(I_0)$.

HAOYUAN: How to more explicitly say the default methods are ok in the new interface table, so that it is sufficient to only prove the generated ofMethod is ok?

By the definition of $\mathsf{ofMethod}(I_0)$, the return value is an object

$$\texttt{return new } I_0\texttt{()\{} \ldots \texttt{\}}$$

To prove it is of type I_0, we use the typing rule (T-OBJ).

HAOYUAN: The order of conditions in (T-OBJ) is not clear below.

- For field typing,

$$\Gamma(\_m_i) = I_i <: I_i$$

- For method typing of getters,

$$\Gamma, m_i : I_i, \texttt{this} : I_0, \Gamma^{mh_i} \vdash m_i \in I_i = I^{mh_i}$$

- For method typing of `with-` methods, by (T-STATICINVK), HAOYUAN: how to do typing for the generated method itself?

$$\Gamma, \overline{m_i : I_i}, \texttt{this} : I_0, \overline{e}_i : \overline{I}_i, \Gamma^{mh_i} \vdash I_0\texttt{.of(}\overline{e}_i\texttt{)} \in I_0 = C^{mh_i}$$

- For method typing of setter methods, HAOYUAN: what about the `with` method?

$$\Gamma, \texttt{this} : I_0, \Gamma^{mh_i} \vdash \texttt{this} \in I_0 = I^{mh_i}$$

- To prove that $\forall i,\ mh_i <: \mathsf{mbody}(m^{mh_i}, I_0)$,
  - For getters,

$$I_i\ m_i\texttt{();} \in \mathsf{fields}(I_0) \Rightarrow I_i\ m_i\texttt{();} \in \mathsf{dom}(I_0)$$

  - For `with-` methods,

$$\begin{array}{l} \mathsf{mbody}(\texttt{with\#}m_i, I_0) \text{ is of form } mh; \\ \mathsf{valid}(I_0) \end{array} \Rightarrow \mathsf{isWith} \Rightarrow I_0 <: \mathsf{mbody}(m^{mh_i}, I_0)$$

- For setters,

$$\begin{array}{l}\mathsf{mbody}(m_i, I_0) \text{ is of form } mh;\\ \mathsf{valid}(I_0)\end{array} \quad \Rightarrow \mathsf{isSetter} \Rightarrow I_0 <: \mathsf{mbody}(m^{mh_i}, I_0)$$

- To prove that such a created object indeed implements all the abstract methods in $\mathsf{dom}(I_0)$, we simply refer to $\mathsf{valid}(I_0)$, since it guarantees each abstract method *meth* to satisfy isField, isWith or isSetter. But that object includes all implementations for those cases, hence it is of type I_0 by (T-OBJ).

Part II. Next we check that in $[\![\mathcal{I}_0]\!]$,

$$\mathsf{dom}([\![\mathcal{I}_0]\!]) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth}) \cup \mathsf{dom}(\overline{meth}')$$

And actually it is guaranteed by **LEMMA 1**.

◀

A.3 Proof of THEOREM

THEOREM. For a given $\mathcal{I}_0 \ldots \mathcal{I}_n$ interface table such that $\forall \mathcal{I} \in \mathcal{I}_0 \ldots \mathcal{I}_n, \mathcal{I}$ OK, then in the interface table $[\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ $\forall \mathcal{I} \in [\![\mathcal{I}_0]\!]\mathcal{I}_1 \ldots \mathcal{I}_n$ either $\mathcal{I}$ OK or $\mathcal{I}$ is a subtype of $\mathcal{I}_0$.

Proof. **LEMMA 2** already proves that $[\![\mathcal{I}_0]\!]$ is OK. On the other hand, if some $\mathcal{I}$ is not a subtype of $\mathcal{I}_0$, the generated code in translation has no way to affect the domain of $\mathcal{I}$, which finishes our proof. ◀