Classless Java

No Author Given

No Institute Given

Abstract.

1 Introduction

– Using annotattions to implement a rich notion of traits with a mechanism to instantiate objects (the of method). Goal 1: is to reduce the amount of code that is required to program with interfaces and default methods. Goal 2: To provide a convenient means to do multiple inheritance in Java.
– Specify the system more formally.
– Show that we can model all trait operations
– Implementation using Lombok
– Case studies: The expression problem, Trivially Case Studies from traits paper.

MARCO: an improvement of this may become part of intro?

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 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.

2 Overview

2.1 A Running Example: Animals

MARCO: fig1 is good but is very big, not sure if it worth its space. If we keep it, we need to do it with a vectorial graphic, or better inside latex (for example with tikz) To propose a standard example, we show `Animals` with a two dimensional

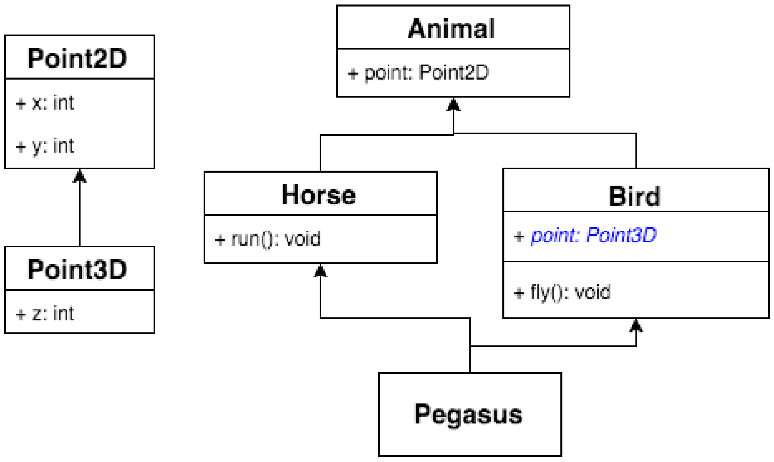

Fig. 1. Pegasus Example.

`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]

`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. 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.

[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.

[2] Available in Java from version ...

By annotating the interface `Point2D`, the annotation will generate 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 `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 equivalent to

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

Notice how 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.

`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.

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

[3] Formal definition of **@Mixin** behaviour is provided later.

```
  }
}
```

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

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

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

Just to give you a feeling on how much boring code **@Mixin** is generating, we show the correspondent code without then **@Mixin** annotation.[4]

```
interface Point3D extends Point2D{
  Point3D withX(int val);
  Point3D withY(int val);
  Point3D withZ(int val);
  public static Point3D of(int x, int y, int 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);}
    }
  }
}
```

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.

`Bird`s are `Animal`s, but while `Animal`s, only need 2D locations, `Bird`s 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...

[4] Can you spot our intentional typo? This kind of code is so boring that our brain refuse to process it. Programmers should not be required to write this kind of code.

– 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 here would think that a language extension is needed. Instead, with our language *restriction* the problem can be easily challenged[5]:

```
@Mixin
interface Bird extends Animal {
  Point3D point();
  void point(Point3D val);
  default void point(Point2D val) {
    if(val instanceof Point3D) { point((Point3D) val); }
    Point3D newPoint = point().withX(val.x()).withY(val.y());
    point(newPoint);
  }
  default void fly() {
    point(point().withX(point().x() + 40));
  }
}
```

Note how

– by covariant method overriding we refine the type of our field
– 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.

The cast is trivially safe since it is guarded by an `instanceof` test. The chain of **`with`**- methods is not fully satisfactory: it requires `Bird` to know of all the fields of `Point2D`. We are planning to extend our **@Mixin** annotation so that we can write `point(point().`**`with`**`(val))`, where the **`with`**`(T val)` method would behave like calling all the **`with`**- where a compliant getter exists in `T`. MARCO: We need to discuss this, we may just implement it? –in this case the text would just become: In Point3D:

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

expandend in

[5] Often in programming languages “freedom is slavery”.

```
interface Point3D extends Point2D{
  ....
  Point3D with(Point2D val){
    if(val instanceof Point3D){return (Point3D)val;}
    return Point3D.of(val.x(),val.y(),this.z());
    }
  ...
  }
```

and here some discussion about the new with method, and then the Bird becomes

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

MARCO: what do we think?

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.

2.2 What @Mixin Generates

Apart from generating code for *getter* methods, the annotation processor can also generate *setter* methods, *with* methods, `clone` methods, etc. We summarize all the forms of method **@Mixin** supports below. Inside the anonymous class in the annotated interface, the following code are generated:

- For methods inside the interface with the form `Tx x()`:
  - `x` is the getter method, with return type `Tx`. Conceptually, it is a member field with name `x` and type `Tx`.
  - generate member field `_x` of type `Tx`, initialized with `x`.
  - generate implemented getter method:

    ```
    public Tx x() { return _x; }
    ```
- For methods inside the interface with the form `void x(Tx x)`:
  - check if exist method `Tx x()`. If not, generate error.

    - generate implemented setter method:

      ```
      public void x(Tx x) { this.x = x; }
      ```

- For methods with the form `T withX(Tx _)`:
    - if there is no `x` field, or type `Tx` does not match, then generate error.
    - implement ‘withX‘ using the ‘of‘ method.
- For methods with the form of `T clone()`: Use `of` method as the constructor, to create a new object with the same field values as the current one.
- For methods with the form of `T x(Tx _)`:
    - check if exist method `T x()`, if not, generate error.
    - inside the inner class, generate

      ```
      public T x(Tx x) { this.x = x; return this;}
      ```

3 Comparing to traits and mixins

Our approach is based on code generation by Java annotations. The model we generate encourages composability and reusability in object-oriented programming, and is considered to be an alternative to traits or mixins, meanwhile achieving better performance in some situations. Hence it is necessary for us to make a comparison between this approach and traits (or mixins) we commonly used before.

Our approach is quite different from mixins, in the sense that we use the trait model of explicitly resolving conflicts. Just as [Scharli2003] demonstrated in the paper, mixin inheritance is a good approach of achieving code reuse, nevertheless, the mixin model is not so expressive to resolve conflicts from many mixins. In the trait model, aliases and exclusions are provided for explicit conflict resolution. Such operations can actually be modelled in the mechanism of our approach. On the other hand, the total ordering property [Scharli2003] restricts the composition of units of behavior. The linearization of mixin inheritance cannot provide a perfect total ordering for various requirements, however, in Java, we have multiple inheritance of interfaces, where methods are inherited at the same time from super interfaces.

Here we present how the original operations on traits are supported by our model.

- **Symmetric sum**: the symmetric composition of two disjoint traits is achieved by simply implementing two interfaces in Java correspondingly, without overriding any method. The composition relies on multiple inheritance on interfaces, which is supported by Java. Below is a simple example.

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

- **Override**: the overriding operation (also known as asymmetric sum) is modelled by implementing many interfaces, while overriding some methods inside. The code below gives an example of explicitly specifying which super interface to refer to, regarding two methods with the same name.

```
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();} }
```

Here the method `m()` in interface `C` simply inherits from `B.m()`.

- **Alias**: an alias operation adds a new name to an old method when creating the new trait. In Java, we just create a new method with reference to the existing method in its super interface. See the example below, where the new method `k()` is an alias of the existing method `m()`.

```
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. See the example below.

```
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.
- We provide `with-` and `clone` methods as auxiliary constructors. A `with`-method is generated for each field, just like record update, returning the new object with that field updated. A `clone` method is generated for the interface, returning a copy of the current object. Furthermore, we do automatic return type refinement for these two kind of methods. This feature is comparatively useful in big examples, making operations and behaviors more flexible, which we will demonstrate later.
- 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.

4 Formal Semantics

MARCO: Old notes: new text is later Yanlin and Haoyuan

We need to show 2 things:

1) The dynamic semantics: what's the code that gets generated by a mixin annotation;

2) The type system: what programs to reject; properties: generation of type-safe/checkable code.

BRUNO: The implementation is still missing the type system (rejecting some programs)!
[6]

e	$::=$	$x \mid e\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid C\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid C\texttt{.}\mathbf{super}\texttt{.}m\texttt{(}\overline{e}\texttt{)} \mid obj$	expressions
obj	$::=$	$\mathbf{new}\ C\texttt{()}\{\overline{field}\ mh_1\{\mathbf{return}\ e_1\texttt{;}\} \ldots mh_n\{\mathbf{return}\ e_n\texttt{;}\}\}$	object creation
$field$	$::=$	$T\ f\texttt{=}x\texttt{;}$	field declaration
I	$::=$	$ann\ \mathbf{interface}\ C_0\ \mathbf{extends}\ C_1 \ldots C_k\{meth_1 \ldots meth_n\}$	interface declaration
$meth$	$::=$	$\mathbf{static}\ mh\ \{\mathbf{return}\ e\texttt{;}\} \mid \mathbf{default}\ mh\{\mathbf{return}\ e\texttt{;}\} \mid mh\texttt{;}$	method declaration
mh	$::=$	$T_0\ m\ \texttt{(}T_1\ x_1 \ldots T_n\ x_n\texttt{)}$	method header
ann	$::=$	$\texttt{@Mixin} \mid \emptyset$	annotations
Γ	$::=$	$x_1{:}C_1 \ldots x_n{:}C_n$	environment

Fig. 2. Grammar of ClassLess Java

(T-INVK)
$$\frac{\forall i \in 0..n\ \Gamma \vdash e_i \in \_ <: C_i \qquad \mathsf{mtype}(m, C_0) = C_1 \ldots C_n \to C}{\Gamma \vdash e_0\texttt{.}m\texttt{(}e_1 \ldots e_n\texttt{)} \in C}$$

(T-STATICINVK)
$$\frac{\forall i \in 1..n\ \Gamma \vdash e_i \in \_ <: C_i \qquad \mathsf{mtypeS}(m, C_0) = C_1 \ldots C_n \to C}{\Gamma \vdash C_0\texttt{.}m\texttt{(}e_1 \ldots e_n\texttt{)} \in C}$$

(T-SUPERINVK)
$$\frac{\Gamma(\mathbf{this}) <: C_0 \qquad \forall i \in 1..n\ \Gamma \vdash e_i \in \_ <: C_i \qquad \mathsf{mtype}(m, C_0) = C_1 \ldots C_n \to C}{\Gamma \vdash C_0\texttt{.}\mathbf{super}\texttt{.}m\texttt{(}e_1 \ldots e_n\texttt{)} \in C}$$

(T-OBJ)
$$\frac{\begin{array}{c}\forall i \in 1..n\ \Gamma, f_1{:}T_1, \ldots, f_k{:}T_k,\ \mathbf{this}{:}C, \Gamma^{mh_i} \vdash e_i \in \_ <: C^{mh_i} \\ \forall i \in 1..n\ mh_i <: \mathsf{mbody}(m^{mh_i}, C) \\ \forall i \in 1..k\ \Gamma(x) = T_i \\ \forall m \text{ such that } \mathsf{mbody}(m, C) = mh; \exists i \in 1..n\ m^{mh_i} = m\end{array}}{\Gamma \vdash \mathbf{new}\ C\texttt{()}\{T_1\ f_1\texttt{=}x_1\texttt{;} \ldots T_k\ f_k\texttt{=}x_k\texttt{;}\ mh_1\{\mathbf{return}\ e_1\texttt{;}\} \ldots mh_n\{\mathbf{return}\ e_n\texttt{;}\}\} \in C}$$

(T-VAR)
$$\frac{\Gamma(x) = C}{\Gamma \vdash x \in C}$$

(T-INTF)
$$\frac{\begin{array}{c}IT(C_0) = ann\ \mathbf{interface}\ C_0\ \mathbf{extends}\ C_1 \ldots C_n\{\overline{meth}\} \\ \forall \mathbf{default}\ mh\{\mathbf{return}\ e\texttt{;}\} \in \overline{meth},\ \Gamma^{mh}, \mathbf{this}{:}C \vdash e \in \_ <: C^{mh} \\ \mathsf{dom}(C_0) = \mathsf{dom}(C_1) \cup \ldots \cup \mathsf{dom}(C_n) \cup \mathsf{dom}(\overline{meth})\end{array}}{I\ \text{OK}}$$

Fig. 3. Typing

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

4.1 Grammar and typing rules

In Figure 2 we show the syntax of ClassLess Java.[7]

We formalize a minimal version of Java 8, focusing on Interfaces, default methods and object creation literals. As you can see, we have no syntax for classes. Expressions are the conventional variable and method call$x$ and$e.m(\overline{e})$, then we have conventional static method call $C.m(\overline{e})$; for simplicity we do not consider the degenerate case of calling a static method over the this receiver as it considered a bad/confusing programming practice. Next we have the more interesting super call $C.\mathtt{super}.m(\overline{e})$, 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 C. Finally, we consider the object initialization expression from an interface C, where (for simplicity) all the fields are initialized with a variable present in scope. Note how our language is exactly a subset of Java 8. We allows the only annotation ... MARCO: later We consider an globally present Interface Table (IT) mapping from interface names C to interface declarations I.

The environment Γ is a mapping from variables to types. As tradition, we allows functional notation for Γ. However, to help us defining auxiliary functions, we also allows functional notation 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 statement $\Gamma \vdash e \in C$ reads "in the environment Γ, expression e has type C." As a shortcut, we write $\Gamma \vdash e \in C <: C'$ instead of $\Gamma \vdash e \in C$ and $C <: C'$. 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 usual, traditional subtyping relation between interfaces.[8]

In Figure 3 we show the typing rules. We use the following auxiliary notation: Γ^{mh} trivially extract the environment from a method header, collect the types and the names of the method parameters. m^{mh} and C^{mh} extracts the method name and the return type from a method header. $\mathsf{mbody}(m, C)$ return the full method declaration as seen by C, that is the method m can be declared in C or inherited from another interface. $\mathsf{mbody}(m, C)$ will be formally defined later.

[7] To be compatible with java, the concrete syntax for an interface declaration with empty supertype list $C_1 \ldots C_n$ would also omit the **extends** keyword.

[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.

$\mathsf{mtype}(m, C)$ and $\mathsf{mtypeS}(m, C)$ return the type signature from a method (using $\mathsf{mbody}(m, C)$ internally). $\mathsf{mtype}(m, C)$ is defined only for non static methods, while $\mathsf{mtypeS}(m, C)$ only on static ones.

We use $\mathsf{dom}(C)$ to denote the set of methods that are defined for type C, that is: $m \in \mathsf{dom}(C)$ iff $\mathsf{mbody}(m, C) = meth$.

We discuss first the most interesting rules, that is (t-Obj) and (t-Intf) MARCO: to complete MARCO: if we have space we can discuss the other rules too, but is not mandatory

4.2 Auxiliary Definitions

Auxiliary function:mbody Defining mbodyis 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. $\mathsf{mbody}(m, C)$ denotes the actual body of method m that interface C owns. It can either be defined originally in C or in its supertypes, and then passed to C 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.

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

As you can see, we are delegating the work to three others auxiliary functions: $\mathsf{tops}(\overline{C}), \mathsf{shadow}(m, \overline{C}) and \mathsf{override}(meth, meth')$
MARCO: we need to discuss the names tops, shadow and override

tops leave only the "needed" superinterfaces, that is, the ones that are not transitivelly reachable by following another superinterface chain. Formally:
$\mathsf{tops}(\overline{C}) = \{\ C \in \overline{C} \mid \nexists C' \in \overline{C} \setminus C,\ C' <: C\ \}$

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 pa-

rameter types. Formally:
$\mathsf{shadow}(m, C_1 \ldots C_n) = \mathsf{shadow}(\overline{meth})$
    where $\mathsf{mbody}(m, C_i) \in \overline{meth}$ if $\mathsf{mbody}(m, C_i) \in \{mh\texttt{;}, \texttt{default}\ mh\texttt{\{return \_;\}}\}$
$\mathsf{shadow}() = \mathsf{None}$
$\mathsf{shadow}(meth) = meth$
$\mathsf{shadow}(\overline{meth}) = \mathsf{mostSpecific}(\overline{meth})$
    where $\overline{meth}$ is of the form $mh_1\texttt{;} \ldots mh_n\texttt{;}$
$\mathsf{shadow}(\overline{meth}) = \mathsf{conflicted} mh;$
    where $\mathsf{mostSpecific}(\overline{meth})) \in \{mh\texttt{;}, \texttt{default}\ mh\texttt{\{return \_;\}}\}$
$\mathsf{mostSpecific}(\overline{meth}) = meth$
    where $meth \in \overline{meth}$ and $\forall meth' \in \overline{meth} : meth <: meth'$,
$T\ m\texttt{(}\, T_1 x_1 \ldots T_n x_n \texttt{)} <: T'\, m\texttt{(}\, T_1 x_1' \ldots T_n x_n' \texttt{)}$
    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 Java ?? 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 $meth' \in \{mh\texttt{;}, \texttt{default}\ mh\texttt{\{return \_;\}}, \mathsf{conflicted}\ mh;\}, meth <: meth'$

YANLIN: for **shadow** conflicted mh;, one missing case is when there are two conflicting default methods with same signature??

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

4.3 Translation Function

$$[\![\texttt{@Mixin interface}\ C_0\ \texttt{extends}\ \overline{C}\{\overline{meth}\}]\!] =$$

$$\emptyset\ \texttt{interface}\ C_0\ \texttt{extends}\ \overline{C}\{\overline{meth};\ \mathsf{ofMethod}(C_0);\ \mathsf{otherMethod}(C_0)\}$$

HAOYUAN: Not $\{\overline{meth}; \ldots\}$, actually. Should avoid duplication.

THEOREM. If $I \longmapsto I'$ and I OK, then I' OK.

Auxiliary function: ofMethod(C_0), otherMethod(C_0)

$$\begin{array}{ll}
\mathsf{ofMethod}(C_0) = \texttt{static}\ C_0\ \texttt{of(}T_1\ m_1\texttt{,}\ \ldots T_k\ m_k\texttt{) \{} & \\
\quad \texttt{return new}\ C_0\texttt{() \{} & \text{if } \mathsf{resolve}(C_0) = \overline{meth} \\
\quad\quad T_1\ \_m_1 = m_1\texttt{;}\ \ldots T_k\ \_m_k = m_k\texttt{;} & \forall meth_i \in \overline{meth}, meth_i = T_i\ m_i\texttt{();} \\
\quad\quad T_i\ m_i\texttt{() \{return}\ \_m_i\texttt{;\}}\ \ldots & T_i\ m_i\texttt{()} \text{ included for all } i \\
\quad\quad C_0\ \texttt{with}m_i\texttt{(}T_i\ x_i\texttt{) \{} & C_0\ \texttt{with}m_i\texttt{(}T_i\ x_i\texttt{)} \text{ included if } \mathsf{with}(meth_i, C_0) \text{ defined} \\
\quad\quad\quad \texttt{return this.of(}m_1\texttt{(),}\ \ldots x_i\texttt{,}\ \ldots m_k\texttt{());} & \\
\quad\quad \texttt{\}}\ \ldots & \\
\quad\quad C_0\ \texttt{clone() \{} & C_0\ \texttt{clone()} \text{ included if } \mathsf{clone}(C_0) \text{ defined} \\
\quad\quad\quad \texttt{return this.of(}m_1\texttt{(),}\ \ldots m_k\texttt{());} & \\
\quad\quad \texttt{\}} & \\
\quad \texttt{\};} & \\
\texttt{\}} & \\
\mathsf{otherMethod}(C_0) = \overline{\mathsf{with}(meth, C_0)}; \mathsf{clone}(C_0) & \text{if } \mathsf{resolve}(C_0) = \overline{meth} \\
\mathsf{resolve}(C_0) = \{\mathsf{mbody}(m, C_0) \mid \mathsf{mbody}(m, C_0)\ \mathsf{isField}\} & \forall m \in \mathsf{dom}(C_0), \text{one case is satisfied:} \\
 & (1)\ \mathsf{mbody}(m, C_0)\ \mathsf{isField} \\
 & (2)\ \mathsf{mbody}(m, C_0)\ \mathsf{isWith} \\
 & (3)\ \mathsf{mbody}(m, C_0)\ \mathsf{isClone} \\
 & (4)\ \mathsf{mbody}(m, C_0)\ \mathsf{isDefault} \\
\mathsf{mbody}(m, C_0)\ \mathsf{isField} & \text{if } \mathsf{mbody}(m, C_0) = T\ m\texttt{();}\ m \neq \texttt{clone} \\
\mathsf{mbody}(m, C_0)\ \mathsf{isWith} & \text{if } mh^{\mathsf{mbody}(m, C_0)} = T'\ \texttt{with}m'\texttt{(}T\ x\texttt{)}, C_0 <: T' \\
 & \text{and } \mathsf{mbody}(m', C_0) = T\ m'\texttt{();}\ m' \neq \texttt{clone} \\
\mathsf{mbody}(m, C_0)\ \mathsf{isClone} & \text{if } mh^{\mathsf{mbody}(m, C_0)} = T\ \texttt{clone()}, C_0 <: T \\
\mathsf{mbody}(m, C_0)\ \mathsf{isDefault} & \text{if } \mathsf{mbody}(m, C_0) = \texttt{default}\ mh\texttt{\{return \_;\}} \\
\mathsf{with}(meth, C_0) = C_0\ \texttt{with}m\texttt{(}T\ x\texttt{);} & \text{if } meth = T\ m\texttt{();}\ \mathsf{mbody}(\texttt{with}m, C_0)\ \mathsf{isWith} \\
\mathsf{clone}(C_0) = C_0\ \texttt{clone();} & \text{if } \mathsf{mbody}(\mathsf{clone}, C_0)\ \mathsf{isClone}
\end{array}$$

Derived notations Below shows how the functions *mtype* and *mmodifier* are derived from *mbody*.

$$\begin{array}{c}
\mathsf{mbody}(m, C) = \mathit{modifier}\ E\ m(\overline{D}\ \overline{x})\{\text{return } e;\} \\
\Rightarrow \mathsf{mtype}(m, C) = \overline{D} \to E,\ \mathsf{mmodifier}(m, C) = \mathit{modifier}
\end{array}$$

5 Implementation

Haoyuan

discuss implementation in lombok; and limitations.

BRUNO: The implementation does not support separate compilation yet. Can we fix this?

6 Case studies

Haoyuan and Yanlin

6.1 A Trivial Solution to the Expression Problem

The *Expression Problem* (EP) [25] 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* [5, 6, 15, 18, 4, 17, 10, 27, 14, 26], or can be used as *design patterns* [9] in existing languages [24, 21, 23, 19, 20]. 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.

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()`.

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 how to add the subtraction variant.

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

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 {
 default String print() {
   return "(" + e1().print() + " + " + e2().print() + ")";
 }
 ExpP e1();
 ExpP e2();
}
```

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 Other case studies

Other case studies using multiple inheritance?

7 Related Work

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

7.1 Multiple Inheritance in Java

Since Java 8 default methods are introduced, which enables concrete method implementation to be defined (via the **default** keyword) inside interface. Since in Java supports extension on multiple interfaces (instead of classes), the introduction of default methods opens the gate of doing multiple inheritance in Java by using interfaces. Details on achieving multiple inheritance by mimicking trait programing through interfaces are summarized by Bono.et. al. [1]. There is also formalization proposed by Goetz and Field [11] to formalize defender (default) methods in Java. However this formalization is limited since they only model exactly one method inside classes/interfaces, where in our formalization multiple methods may be defined.

YANLIN: need to discuss: whether the comments for FTJ is fair. There are proposals for extending Java with traits. For example, FeatherTrait Java (FTJ) [13] by Liquori et al. extends the calculus of Featherweight Java (FJ) [12] with statically-typed traits, adding trait-based inheritance in Java. However, language extensions (including FTJ) have a natural drawback: the programmer has to learn new syntax. In contrast, our approach is completely compatible with the current Java language, so that programmers don't need to pay any learning cost to adapt to this new classless programming pattern. Another drawback which is particular for FTJ is that FTJ doesn't have type for traits, hence the correctness check of trait is done when type-checking classes. This choice makes the design of FTJ simpler but lost typechecking efficiency (one trait will be potentially checked multiple times if it is used in multiple classes).

7.2 Multiple Inheritance in General

Multiple inheritance is a great feature to have in a programming language, but it is difficult to model and implement. These difficulties include the famous diamond (fork-join) problem, conflicting methods, yo-yo problem, etc. In order to model multiple inheritance, lots of models have been proposed in the past few years, including mixins [2] and traits [22]. Both traits and mixins provide novel models of new programming architecture in object oriented programming paradigm.

1) Mixins model named components that can be applied to various classes as reusable functionality units. However, the well-known limitation of mixins is that they suffer from linearization constraints, which hinders the expressiveness of mixins and the ability of resolving conflicts. As shown in section 3, although they solved some multiple inheritance problems, they are extremely restricting expressiveness.

2) Traits are a novel model to differentiate two functionalities out of classes: units of reusable components and object factories. In trait models, traits are responsible for units of reusable components, which contain pure methods as reusable functionalities; whereas classes act as object factories, which requires various functionalities from multiple traits. However drawbacks of traits are obvious: traits do not allow state; traits can't be instantiated directly.

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) [7], however suffers from object initialization problem as pointed out by Malayeri et al. [16]. It bypasses all constructor calls to virtual superclasses, which would potentially cause serious semantic error. Scala solution (traits) avoids this problem by disallowing constructor parameters, causing no ambiguity in cases such as diamond problem. The drawback of this elegant design is limited expressiveness.

ThisType/MyType/Extensibility Typical object-oriented programming languages (e.g. Java) use class names as types and inheritance as subtyping relations. This gives difficulty when extending classes/interfaces with recursive types. Problems such as binary methods, return type refinement, etc. emerge and hinders software extensibility greatly.

MyType [3] was proposed by Bruce and Foster, which represents the type of **this** and changes its meaning along with inheritance. The addition of MyType to a language will allow easy definition of binary methods, methods with recursive types (i.e., the same type of the receiver appears in the argument or return positions of methods), etc. MyType greatly enhances the expressiveness and extensibility of object-oriented programming languages.

In our approach, we are using covariant-return types to simulate some uses of MyType. But our approach only works on method positive positions, whereas MyType is more general, as it works on any positions. Nevertheless our approach is still useful for modeling fluent interfaces and solving expression problems,etc.

Type-Directed Translations/Syntactic Sugar - SugarJ [8]

We can model certain types of language extensions with annotations only, but those extensions do not introduce new syntax: they merely do automatic code generation.

Meta-programming techniques - MetaFJig*

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?

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

8 Conclusion

References

1. Bono, V., Mensa, E., Naddeo, M.: 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)
2. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP '90 (1990)
3. Bruce, K.B.: A paradigmatic object-oriented programming language: Design, static typing and semantics. Journal of Functional Programming 4(02), 127–206 (1994)
4. Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ECOOP'98 (1998)
5. Chambers, C., Leavens, G.T.: Typechecking and modules for multimethods. ACM Trans. Program. Lang. Syst. 17, 805–843 (November 1995)
6. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: modular open classes and symmetric multiple dispatch for java. In: OOPSLA '00 (2000)

7. Ellis, M.A., Stroustrup, B.: The annotated C++ reference manual. Addison-Wesley (1990)
8. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: Sugarj: Library-based syntactic language extensibility. OOPSLA'11 (2011)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
10. Garrigue, J.: Programming with polymorphic variants. In: ML Workshop (1998)
11. Goetz, B., Field, R.: Featherweight defenders: A formal model for virtual extension methods in java. `http://cr.openjdk.java.net/âĹijbriangoetz/lambda/featherweight-defenders.pdf` (2012)
12. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: A minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. 23(3) (May 2001)
13. Liquori, L., Spiwack, A.: Feathertrait: A modest extension of featherweight java. ACM Trans. Program. Lang. Syst. 30(2) (Mar 2008)
14. Löh, A., Hinze, R.: Open data types and open functions. In: PPDP '06 (2006)
15. Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA '89 (1989)
16. Malayeri, D., Aldrich, J.: CZ: multiple inheritance without diamonds, vol. 44. ACM (2009)
17. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: new-age components for old-fashioned java. In: OOPSLA '01 (2001)
18. Nystrom, N., Qi, X., Myers, A.C.: J&: nested intersection for scalable software composition. In: OOPSLA '06 (2006)
19. Oliveira, B.C.d.S.: Modular visitor components. In: ECOOP'09 (2009)
20. Oliveira, B.C.d.S., Cook, W.R.: Extensibility for the masses: Practical extensibility with object algebras. In: ECOOP'12 (2012)
21. Oliveira, B.C.d.S., Hinze, R., Löh, A.: Extensible and modular generics for the masses. In: Trends in Functional Programming (2006)
22. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP'03 (2003)
23. Swierstra, W.: Data types à la carte. Journal of Functional Programming 18(4), 423 – 436 (2008)
24. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: ECOOP'04 (2004)
25. Wadler, P.: The Expression Problem. Email (Nov 1998), discussion on the Java Genericity mailing list
26. Wehr, S., Thiemann, P.: JavaGI: The interaction of type classes with interfaces and inheritance. ACM Trans. Program. Lang. Syst. 33 (July 2011)
27. Zenger, M., Odersky, M.: Extensible algebraic datatypes with defaults. In: ICFP '01 (2001)