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.

2 Overview

Yanlin
* Explain what the Mixin annotations do using examples.
* Motivate the use of multiple inheritance in Java.
* Maybe use Marco's example (Point example).

We provide a Java annotation **@Mixin**to provide default implementations for various methods and a mechanism to instantiate objects. **@Mixin**annotation helps programmers to write less cumbersome code and instantiate interfaces in Java.

For example, interface `Point` annotated with **@Mixin**:

```
@Mixin
interface Point {
   int X();
   int Y();
   void X(int x);
   void Y(int y);
   Point withX(int x);
   Point withY(int y);
   Point clone();
   default int distance() {
      return (int) Math.sqrt(X() * X() + Y() * Y());
   }
}
```

`Point` has two (conceptually) member fields `X` and `Y`. Methods `int X()` and `int Y()` serve as *getter* methods. Methods `void X(int X)` and `void Y(int Y)` serve as *setter* methods. Method `Point withX(int X)` updates field `X` and returns **this**.

A typical and trivial implementation that programmers usually do is:

```
class PointImpl implements Point {
   private int _X;
   private int _Y;
   public PointImpl(int X, int Y) {
      this._X = X;
      this._Y = Y;
   }
   public int X() {
      return _X;
   }
   public int Y() {
      return _Y;
   }
   public Point withX(int X) {
      X(X);
      return this;
   }
   public void X(int X) {
      _X = X;
   }
   public void Y(int Y) {
      _Y = Y;
   }
   public Point withY(int Y) {
      Y(Y);
      return this;
   }
   public Point clone() {
      return new PointImpl(_X, _Y);
   }
}
```

`PointImpl` implements `Point` and provides a constructor with quite mechanical code. What's worse, the implementation in `PointImpl` may not be reused in a single inheritance language. However, with our approach, the **@Mixin**annotation will generate a static method `of` inside `Point`. `of` makes use of Java anonymous classes and achieves the same implementation as `PointImpl`.

```
   // inside interface Point
   static Point of(int X, int Y) {
      return new Point() {
         int _X = X;
         public int X() {
            return _X;
         }
         int _Y = Y;
         public int Y() {
            return _Y;
```

```
        }
        public Point withX(int X) {
           return of(X, Y());
        }
        public void X(int X) {
           _X = X;
        }
        public void Y(int Y) {
           _Y = Y;
        }
        public Point withY(int Y) {
           return of(X(), Y);
        }
        public Point clone() {
           return of(X(), Y());
        }
     };
  }
```

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:

    ```
    pubic 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

    ```
    pubic 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.

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.
- **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 `of` methods in our model, as a replacement to the constructors in original traits. Furthermore, we also support `with` and `clone` methods as auxiliary constructors, making the creation of instances more flexible and convenient. Conversely, there are certain operations we cannot model, such as method renaming (as in [Reppy2006]), which breaks structural subtyping.

A further feature leads to return type refinement in our model. Generally speaking, when we use inheritance to create a new trait, with the return type of an existing method being refined, the new `of` method keeps this consistency. This feature is very useful in many applications; we will see how it makes a difference in our Expression Problem example, in Section [Case Study].

4 Formal Semantics

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)!

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

6.2 Other case studies

BRUNO: The case studies still need to be implemented!

Collections example from traits paper? (YANLIN: couldn't found source code)

Other case studies using multiple inheritance?

7 Related Work

- traits (original, variations, scala) - mixins (original, scala) - multiple inheritance - expression problem - ...

8 Conclusion

References