FHJ: A Formal Model for Hierarchical Dispatching and Overriding

Yanlin Wang , Haoyuan Zhang , Bruno C. d. S. Oliveira , Marco Servetto

Motivation

1. In OOP, multiple inheritance is hard due to the ambiguity arising from inheriting multiple parents (with conflicting methods).
2. Numerous existing work provides solutions for conflicts which arise from *diamond inheritance*.
3. However, these solutions are inadequate to deal with *unintentional method conflicts*: conflicts which arise from two unrelated methods that happen to share the same signature.

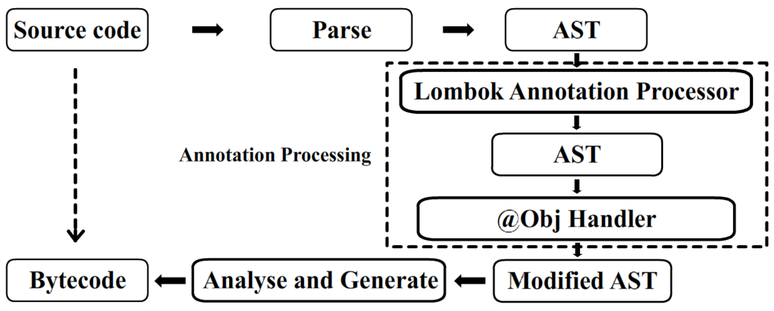
Contributions

1. Abstract state operations:
   - a new way to think about state, via operations instead of fields directly
   - easy to combine with *multiple (trait) inheritance*
   - supports constructors, co-variant type-refinement of state
2. Classless Java: a practical realization of IB in Java.

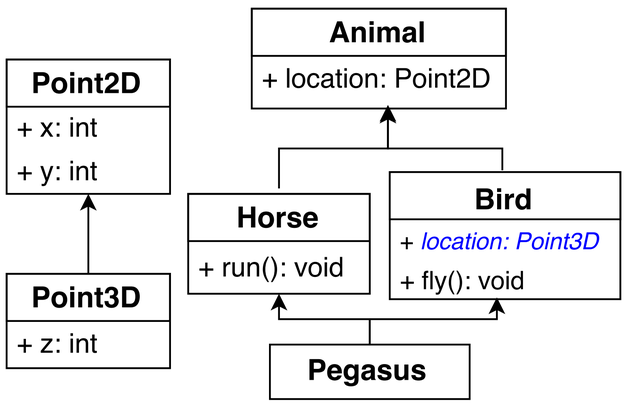
Implementation

Java supports compilation agents, where Java libraries can interact with the Java compilation process, acting as a man in the middle between the generation of AST and bytecode.

This process is facilitated by frameworks like Lombok: a Java library that aims at reducing Java boilerplate code via annotations. `@Obj` was created using Lombok.



UML Diagram



Results (Partial)

In the maze game case study, both SLOC and # of interfaces are greatly reduced:

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

Object Interfaces and Instantiation

```
@Obj interface Horse extends Animal {
  default void run() {out.println("running!");} }
@Obj interface Bird extends Animal {
  default void fly() {out.println("flying!");} }
@Obj interface Pegasus extends Horse, Bird {}
```

```
Pegasus p = Pegasus.of();
interface Pegasus extends Horse, Bird {
  // generated code not visible to users
  static Pegasus of() { return new Pegasus() {}; }
}
```

Object Interfaces with State (immutable data)

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

```
@Obj interface Point2D { int x(); int y(); }
Point2D myPoint = Point2D.of(4, 2);
Point2D p = Point2D.of(42, myPoint.y());
```

with- Methods

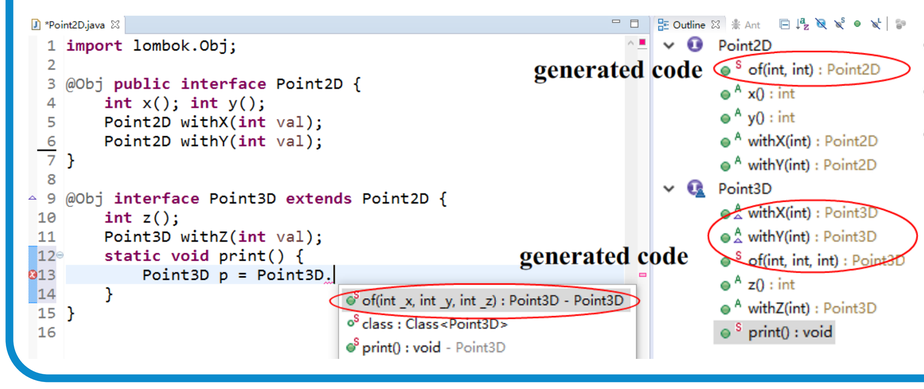
```
@Obj interface Point2D {
   int x(); int y(); // getters
   // with- methods
   Point2D withX(int val); Point2D withY(int val); }
Point2D p = myPoint.withX(42);
```

Mutable Data & Field Type Refinement

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

More in the paper

	Operation	Example	Description
State operations (for a field x)	**"fields"/getters**	`int x()`	Retrieves value from field x.
	withers	`Point2D withX(int val)`	Clones object; updates field x to val.
	setters	`void x(int val)`	Sets the field x to a new value val.
	fluent setters	`Point2D x(int val)`	Sets the field x to val and returns this.
Other operations	**factory methods**	`static Point2D of(int _x,int _y)`	Factory method (generated).
	functional updaters	`Point3D with(Point2D val)`	Updates all matching fields in val.



- Techniques for field type refinement.
- Formalization and proofs.
- Case studies and applications.
  [The Expression Problem]
  [Embedded DSLs with Fluent Interfaces]
  [A Maze Game]
  [Refactoring an Interpreter]

Acknowledgments

This work is sponsored by the Hong Kong Research Grant Council Early Career Scheme project number 27200514. The first and second authors are supported by the SIG PAC student travel grant.

Contacts

The University of Hong Kong
Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira: *{ylwang,hyzhang,bruno}@cs.hku.hk*

Victoria University of Wellington
Marco Servetto: *marco.servetto@ecs.vuw.ac.nz*