FHJ: A Formal Model for Hierarchical Dispatching and Overriding

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

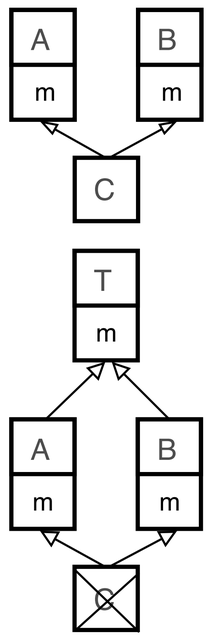
Motivation

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

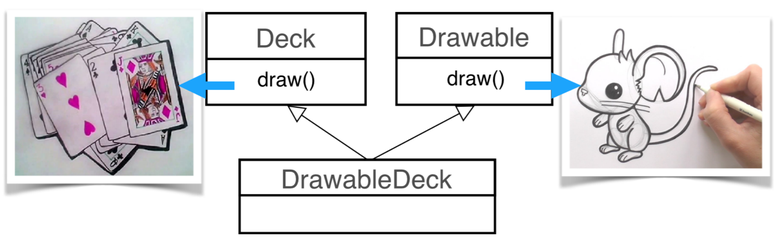
1. A formalization of the hierarchical dispatching algorithm
2. Hierarchical overriding: override individual branches of the class hierarchy.
3. FHJ: a formalized model based on Featherweight Java
4. Prototype implementation: FHJ interpreter in Scala.

Real Conflicts



1. Real conflicts needs to be explicitly resolved by programmers.
2. Existing OOP models have taken care of this case intensively.

Unintentional Conflicts



1. Conflicting methods have completely different meaning/domain which just share the same name (and parameter types).
2. Few existing OOP models support unintentional method conflicts well.

P1: basic unintentional method conflicts

```
interface Deck { void draw() {...} }
interface Drawable { void draw() {...} }
interface DrawableDeck extends Drawable, Deck {}
// main program
((Deck) new DrawableDeck()).draw() // calls Deck.draw
// new DrawableDeck().draw() // ambiguous!!!
```

P2: dynamic dispatching

```
interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}
interface Drawable {...}
interface SafeDeck extends Deck {...}
interface DrawableSafeDeck extends Drawable, SafeDeck {}
new DrawableSafeDeck().shuffleAndDraw() // SafeDeck.draw is called
```

P3: overriding on individual branches

```
interface DrawableSafeDeck extends Drawable, SafeDeck {
  void draw() override Drawable {
    JFrame frame = new JFrame("Canvas");
    frame.setSize(600, 600);
    frame.getContentPane().setBackground(Color.red);
    frame.getContentPane().add(new Square(10,10,100,100)); ...
  }
}
((Drawable)new DrawableSafeDeck()).draw();
//calls the draw in DrawableSafeDeck
```

Syntax

Interfaces	IL	::=	`interface I extends` $\overline{\mathrm{I}}$ $\{\overline{\mathrm{M}}\}$
Methods	M	::=	I m($\overline{\mathrm{I_x}}$ $\overline{\mathrm{x}}$) `override` J $\{$`return` e;$\}$ | I m($\overline{\mathrm{I_x}}$ $\overline{\mathrm{x}}$) `override` J ;
Expressions	e	::=	x | e.m($\overline{e}$) | `new` I() | (I)e
Context	Γ	::=	$\overline{\mathrm{x}} : \overline{\mathrm{I}}$
Values	v	::=	(I)`new` J()

The Key Semantics Rule

$$\text{(S-Invk)}\ \frac{\texttt{mbody}(\mathrm{m}, \mathrm{I}, \mathrm{J}) = (\mathrm{I}_0, \overline{\mathrm{I_x}}\ \overline{\mathrm{x}}, \mathrm{I}_e\ e_0)}{((\mathrm{J})\texttt{new}\ \mathrm{I}())\,.\mathrm{m}(\overline{v}) \rightarrow (\mathrm{I}_e)[(\mathrm{I_x})v/\overline{\mathrm{x}}, (\mathrm{I}_0)\texttt{new}\ \mathrm{I}()/\texttt{this}]e_0}$$

static type — dynamic type — paras substitution — this substitution

mbody(m, I, J)

[tracks the path info in hierarchical dispatch]

static type of e0

More in the paper

- Intuitive examples
- Subtyping, typing and reduction rules
- Design choices
- Extensions
- Properties/Proofs
- ...

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*