

FHJ: A Formal Model for Hierarchical Dispatching and Overriding

Yanlin Wang

The University of Hong Kong, China

ylwang@cs.hku.hk

Haoyuan Zhang

The University of Hong Kong, China

hyzhang@cs.hku.hk

Bruno C. d. S. Oliveira

The University of Hong Kong, China

bruno@cs.hku.hk

Marco Servetto

Victoria University of Wellington, New Zealand

marco.servetto@ecs.vuw.ac.nz

Abstract

Multiple inheritance is a valuable feature for Object-Oriented Programming. However, it is also tricky to get right, as illustrated by the extensive literature on the topic. A key issue is the *ambiguity* arising from inheriting multiple parents, which can have conflicting methods. Numerous existing work provides solutions for conflicts which arise from *diamond inheritance*: i.e. conflicts that arise from implementations sharing a common ancestor. However, most mechanisms are inadequate to deal with *unintentional method conflicts*: conflicts which arise from two unrelated methods that happen to share the same name and signature.

This paper presents a new model called *Featherweight Hierarchical Java* (**FHJ**) that deals with unintentional method conflicts. In our new model, which is partly inspired by C++, conflicting methods arising from unrelated methods can coexist in the same class, and *hierarchical dispatching* supports unambiguous lookups in the presence of such conflicting methods. To avoid ambiguity, hierarchical information is employed in method dispatching, which uses a combination of static and dynamic type information to choose the implementation of a method at run-time. Furthermore, unlike all existing inheritance models, our model supports *hierarchical method overriding*: that is, methods can be *independently overridden* along the multiple inheritance hierarchy. We give illustrative examples of our language and features and formalize **FHJ** as a minimal Featherweight-Java style calculus.

2012 ACM Subject Classification Dummy classification

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.20

Acknowledgements I want to thank ...

1 Introduction

Inheritance in Object-Oriented Programming (OOP) offers a mechanism for code reuse. However many OOP languages are restricted to single inheritance, which is less expressive and flexible than multiple inheritance. Nevertheless, different flavours of multiple inheritance have been adopted in some popular OOP languages. C++ has had multiple inheritance from



© John Q. Public and Joan R. Public;

licensed under Creative Commons License CC-BY

32nd European Conference on Object-Oriented Programming (ECOOP 2018).

Editor: Todd Millstein; Article No. 20; pp. 20:1–20:30

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the start. Scala adapts the ideas from traits [28, 9, 16] and mixins [5, 11, 32, 2, 13] to offer a disciplined form of multiple inheritance. Java 8 offers a simple variant of traits, disguised as interfaces with default methods [12].

A reason why programming languages have resisted to multiple inheritance in the past is that, as Cook [7] puts it, “*multiple inheritance is good but there is no good way to do it*”. One of the most sensitive and critical issues is perhaps the ambiguity introduced by multiple inheritance. One case is the famous *diamond problem* [27, 29] (also known as *fork-join inheritance* [27]). In the diamond problem, inheritance allows one feature to be inherited from multiple parent classes that share a common ancestor. Hence conflicts arise. The variety of strategies for resolving such conflicts urges the occurrence of different multiple inheritance models, including traits, mixins, CZ [17], and many others. Existing languages and research have addressed the issue of diamond inheritance extensively. Other issues including how multiple inheritance deals with state, have also been discussed quite extensively [33, 17, 31].

In contrast to diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [28]. In this case, conflicting methods do not actually refer to the same feature. In a nominal system, methods can be designed for different functionality, but happen to have the same names (and signatures). A simple example of this situation is two `draw` methods that are inherited from a deck of cards and a drawable widget, respectively. In such context, the two `draw` methods have very different meanings, but they happen to share the same name. When inheritance is used to compose these classes, a compilation error happens due to conflicts. However, unlike the diamond problem, the conflicting methods have very different meanings and do not share a common parent. We call such a case *fork inheritance*, in analogy to diamond inheritance.

When unintentional method conflicts happen, they can have severe effects in practice if no appropriate mechanisms to deal with them are available. In practice, existing languages only provide limited support for the issue. In most languages, the mechanisms available to deal with this problem are the same as the diamond inheritance. However, this is often inadequate and can lead to tricky problems in practice. This is especially the case when it is necessary to combine two large modules and their features, but the inheritance is simply prohibited by a small conflict. As a workaround from the diamond inheritance side, it is possible to define a new method in the child class to override those conflicting methods. However, using one method to fuse two unrelated features is clearly unsatisfactory. Therefore we need a better solution to keep both features separately during inheritance, so as not to break *independent extensibility* [36].

C++ and C# do allow for two unintentionally conflicting methods to coexist in a class. C# allows this by interface multiple inheritance and explicit method implementations. But since C# is a single inheritance language, it is only possible to *implement* multiple interfaces (but not multiple classes). C++ accepts fork inheritance and resolves the ambiguity by specifying the expected path by *upcasts*. However, neither the C# nor C++ approaches allow such conflicting methods to be further overridden. Some other workarounds or approaches include delegation and renaming/exclusion in the trait model. However, renaming/exclusion can break the subtyping relation between a subclass and its parent. This is not adequate for the class model commonly used in mainstream OOP languages, where the subclass is always expected to be a subtype of the parent class.

This paper proposes two mechanisms to deal with unintentional method conflicts: *hierarchical dispatching* and *hierarchical overriding*. Hierarchical dispatching is inspired by the mechanisms in C++ and provides an approach to method dispatching, which combines static and dynamic information. Using hierarchical dispatching, the method binder will



■ **Figure 1** DrawableSafeDeck: an illustration of hierarchical overriding.

look at both the *static type* and the *dynamic type* of the receiver during runtime. When there are multiple branches that cause unintentional conflicts, the static type can specify one branch among them for unambiguity, and the dynamic type helps to find the most specific implementation. In that case, both unambiguity and extensibility are preserved. The main novelty over existing work is the formalization of the essence of a hierarchical dispatching algorithm, which (as far as we know) has not been formalized before.

Hierarchical overriding is a novel language mechanism that allows method overriding to be applied only to one branch of the class hierarchy. Hierarchical overriding adds expressive power that is not available in languages such as C++ or C#. Hierarchical overriding allows overriding to work for classes with multiple (conflicting) methods sharing the same names and signatures. An example is presented in Figure 1. In this example, there are 4 classes/interfaces. Two classes `Deck` and `Drawable` model a deck of cards and a drawable widget, respectively. The class `SafeDeck` adds functionality to check whether the deck is empty so as to prevent drawing a card from an empty deck. The interesting class is `DrawableSafeDeck`, which inherits from both `SafeDeck` and `Drawable`. Hierarchical overriding is used in `DrawableSafeDeck` to keep two separate `draw` methods for each parent, but override *only* the `draw` method coming from `Drawable`, in order to draw a widget with a deck of cards. Note that hierarchical overriding is denoted in the UML diagram with the notation `draw()↑Drawable`, expressing that the `draw` method from `Drawable` is overridden. Although in this example only one of the `draw` methods is overridden (and the other is simply inherited), hierarchical overriding supports multiple conflicting methods to be independently overridden as well.

To present hierarchical overriding and dispatching, we introduce a formalized model **FHJ** in Section 3 based on Featherweight Java [14], together with theorems and proofs for type soundness. We also have a prototype implementation of a **FHJ** interpreter written in Scala. The implementation validates all the examples presented in the paper. One nice feature of the implementation is that it can show the detailed step-by-step evaluation of the program, which is convenient for understanding and debugging programs & semantics.

In summary, our contributions are:

- **A formalization of the hierarchical dispatching algorithm** that integrates both the static type and dynamic type for method dispatch, and ensures unambiguity as well as extensibility in the presence of unintentional method conflicts.
- **Hierarchical overriding**: a novel notion that allows methods to override individual branches of the class hierarchy.
- **FHJ**: a formalized model based on Featherweight Java, supporting the above features. We provide the static and dynamic semantics and prove the type soundness of the model.

- 125 ■ **Prototype implementation**¹: a simple implementation of **FHJ** interpreter in Scala.
- 126 The implementation can type-check and run variants of all the examples shown in this
- 127 paper.

128 2 A Running Example: Drawable Deck

129 This section illustrates the problem of unintentional method conflicts, together with the
 130 features of our model for addressing this issue, by a simple running example. In the
 131 following text we will introduce three problems one by one and have a discussion on possible
 132 workarounds and our solutions. Problems 1 and 2 are related to hierarchical dispatching, and
 133 in C++ it is possible to have similar solutions to both problems. Hence it is important to
 134 emphasize that, with respect to hierarchical dispatching, our model is not a novel mechanism.
 135 Instead, inspired by the C++ solutions, our contribution is formalizing a minimal calculus of
 136 this feature together with a proof of type soundness. However, for the final problem, there
 137 is no satisfactory approach in existing languages, thus what we propose is a novel feature
 138 (hierarchical overriding) with the corresponding formalization of that feature.

139 In the rest of the paper, we use a Java-like syntax for programs. All types are defined
 140 with the keyword **interface**; the concept is closely related to Java 8 interfaces with default
 141 methods [4] and traits. In short, an interface in our model has the following characteristics:

- 142 ■ It allows multiple inheritance.
- 143 ■ Every method is either abstract or implemented with a body (like Java 8 default methods).
- 144 ■ The **new** keyword is used to instantiate an interface.
- 145 ■ It cannot have state.

146 2.1 Problem 1: Basic Unintentional Method Conflicts

147 Suppose that two components **Deck** and **Drawable** have been developed in a system. **Deck**
 148 represents a deck of cards and defines a method **draw** for drawing a card from the deck.
 149 **Drawable** is an interface for graphics that can be drawn and also includes a method called
 150 **draw** for visual display. For simple illustration, the default implementation of **draw** in **Drawable**
 151 only creates a blank canvas on the screen, while the **draw** method in **Deck** simply prints out a
 152 message "Draw a card.".

```

153 interface Deck {
154     void draw() { // draws a card from the Deck
155         println("Draw a card.");
156     }
157 }
158
159 interface Drawable {
160     void draw() { // create a blank canvas
161         JFrame frame = new JFrame();
162         frame.setVisible(true);
163     }
164 }
165 
```

166 In **Deck**, **draw** uses **println**, which is a library function. The two **draw** methods can have
 167 different return types, but for simplicity, the return types are both **void** here. Note that,
 168 similarly to Featherweight Java [14], **void** is unsupported in our formalization. We could have
 169 also defined an interface called **Void** and return an object of that type instead. To be concise,

¹ The implementation is available at <https://github.com/YanlinWang/MIM/tree/master/Calculus>

however, we use **void** in our examples. In interface **Drawable**, the **draw** method creates a blank canvas.

Now, suppose that a programmer is designing a card game with a GUI. He may want to draw a deck on the screen, so he first defines a drawable deck using multiple inheritance:

```
interface DrawableDeck extends Drawable, Deck {}
```

The point of using multiple inheritance is to compose the features from various components and to achieve code reuse, as supported by many mainstream OO languages. Nevertheless, at this point, languages like Java simply treat the two **draw** methods as the same, hence the compiler fails to compile the program and reports an error.

This case is an example of a so-called *unintentional method conflict*. It arises when two inherited methods happen to have the same name and parameter types, but they are designed for different functionalities with different semantics. Now one may quickly come up with a workaround, which is to manually merge the two methods by creating a new **draw** method in **DrawableDeck** to override the old ones. However, merging two methods with totally different functionalities does not make any sense. This non-solution would hide the old methods and break independent extensibility.

2.1.1 Problem and Possible Workarounds

The essential problem is how to resolve unintentional method conflicts and invoke the conflicting methods separately without ambiguity. To tackle this problem, there are several other workarounds that come to our mind. We briefly discuss those potential fixes and workarounds next:

- *I. Delegation.* As an alternative to multiple inheritance, delegation can be used by introducing two fields (or field methods) with the **Drawable** type and **Deck** type, respectively. Although it avoids method conflicts, it is known that using delegation makes it hard to correctly maintain self-references in an extensible system and also introduces a lot of boilerplate code.
- *II. Refactor **Drawable** and/or **Deck** to rename the methods.* If the source code for **Drawable** or **Deck** is available then it may be possible to rename one of the **draw** methods. However this approach is non-modular, as it requires modifying existing code and becomes impossible if the code is unavailable.
- *III. Method exclusion/renaming.* Eiffel [18] and some trait models support method exclusion/renaming. Those features can eliminate conflicts, although most programming languages do not support them. In a traditional OO system, they can break the subtyping relationship. Moreover, in contrast with exclusion, renaming can indeed preserve both conflicting behaviours. However, it is cumbersome in practice, as introducing new names can affect other code blocks.

2.1.2 FHJ's solution

To solve this problem it is important to preserve both conflicting methods during inheritance instead of merging them into a single method. Therefore **FHJ** accepts the definition of **DrawableDeck**. To disambiguate method calls, we can use *upcasts* in **FHJ** to specify the “branch” in the inheritance hierarchy that should be called. The following code illustrates the use of upcasts for disambiguation:

```
interface Deck { void draw() {...} }
interface Drawable { void draw() {...} }
```

```

217 interface DrawableDeck extends Drawable, Deck {}
218 // main program
219 ((Deck) new DrawableDeck()).draw() // calls Deck.draw
220 // new DrawableDeck().draw() // this call is ambiguous and rejected
221

```

222 **YANLIN: Refined.** In our language, a program consists of interfaces declarations and a main
 223 an expression which produces the final result. In the above main expression `((Deck)new`
 224 `DrawableDeck()).draw()`, the cast indicates that we expect to invoke the `draw` method from
 225 the branch `Deck`. Similarly, we could have used an upcast to `Drawable` to call the `draw` method
 226 from `Drawable`. Without the cast, the call would be ambiguous and **FHJ**'s type system would
 227 reject it.

228 This example illustrates the basic form of fork inheritance, where two unintentionally
 229 conflicting methods are accepted by multiple inheritance. Note that C++ supports this
 230 feature and also addresses the ambiguity by upcasts. The code for the above example in
 231 C++ is similar.

232 2.2 Problem 2: Dynamic Dispatching

233 Using explicit upcasts for disambiguation helps when making calls to classes with conflicting
 234 methods, but things become more complicated with dynamic dispatching. Dynamic dispatch-
 235 ing is very common in OO programming for code reuse. Let us expand the previous example
 236 a bit, by redefining those interfaces with more features:

```

237 interface Deck {
238   void draw() {...}
239   void shuffle() {...}
240   void shuffleAndDraw() { this.shuffle(); this.draw(); }
241 }
242
243

```

244 Here `shuffleAndDraw` invokes `draw` from its own enclosing type. In **FHJ**, this invocation is
 245 dynamically dispatched. This is important, because a programmer may define a subtype of
 246 `Deck` and override the method `draw`:

```

247 interface SafeDeck extends Deck {
248   boolean isEmpty() {...}
249   void draw() { // overriding
250     if (isEmpty()) println("The deck is empty.");
251     else println("Draw a card");
252   }
253 }
254
255

```

256 Without dynamic dispatching, we may have to copy the `shuffleAndDraw` code into `SafeDeck`, so
 257 that `shuffleAndDraw` calls the new `draw` defined in `SafeDeck`. Dynamic dispatching immediately
 258 saves us from the duplication work, since the method becomes automatically dispatched
 259 to the most specific one. Nevertheless, as seen before, dynamic dispatch would potentially
 260 introduce ambiguity. For instance, when we have the class hierarchy structure shown in
 261 Figure 2(left) with the following code:

```

262 interface DrawableSafeDeck extends Drawable, SafeDeck {}
263 // main program
264 new DrawableSafeDeck().shuffleAndDraw()
265
266

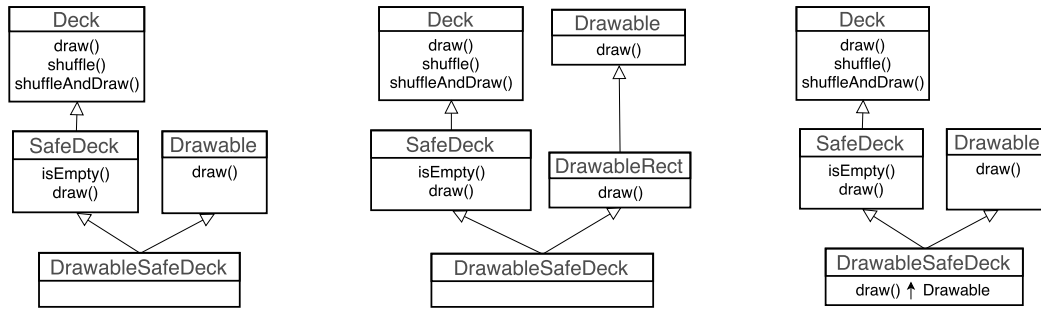
```

267 Indeed, using reduction steps following the reduction rules in FJ [14]-like languages, where
 268 no static types are tracked, the reduction steps would roughly be:

```

269 new DrawableSafeDeck().shuffleAndDraw()
270 -> new DrawableSafeDeck().shuffle(); new DrawableSafeDeck().draw()
271 -> ...
272

```



■ **Figure 2** UML diagrams for 3 variants of DrawableSafeDeck.

```

273 -> new DrawableSafeDeck().draw()
274 -> <<error: ambiguous call!!!>>
275

```

276 When the DrawableSafeDeck object calls shuffleAndDraw, the implementation in Deck is dis-
 277 patched. But then shuffleAndDraw invokes “this.draw()”, and at this point, the receiver is
 278 replaced by the object **new** DrawableSafeDeck(). From the perspective of DrawableSafeDeck,
 279 the draw method seems to be ambiguous since DrawableSafeDeck inherits two draw meth-
 280 ods from both SafeDeck and Drawable. But ideally we would like shuffleAndDraw to invoke
 281 SafeDeck.draw because they belong to the same class hierarchy branch.

2.2.1 FHJ’s solution

282 The essential problem is how to ensure that the correct method is invoked. To solve this
 283 problem, **FHJ** uses a variant of method dispatching that we call *hierarchical dispatching*. In
 284 hierarchical dispatching, both the static and dynamic type information are used to select
 285 the right method implementation. During runtime, a method call makes use of both the
 286 static type and the dynamic type of the receiver, so it is a combination of static and dynamic
 287 dispatching. Intuitively, the static type specifies one branch to avoid ambiguity, and the
 288 dynamic type finds the most specific implementation on that branch. To be specific, the
 289 following code is accepted by **FHJ**:

```

290 interface Deck {
291   void draw() {...}
292   void shuffle() {...}
293   void shuffleAndDraw() { this.shuffle(); this.draw(); }
294 }
295 interface Drawable {...}
296 interface SafeDeck extends Deck {...}
297 interface DrawableSafeDeck extends Drawable, SafeDeck {}
298 // main program
299 new DrawableSafeDeck().shuffleAndDraw() // SafeDeck.draw is called
300

```

303 The computation performed in **FHJ** is as follows:

```

304   new DrawableSafeDeck().shuffleAndDraw()
305 -> ((DrawableSafeDeck) new DrawableSafeDeck()).shuffleAndDraw()
306 -> ((Deck) new DrawableSafeDeck()).shuffle(); ((Deck) new DrawableSafeDeck()).draw()
307 -> ...
308 -> ((Deck) new DrawableSafeDeck()).draw()
309 -> ... // SafeDeck.draw
310

```

312 Notably, we track the static types by adding upcasts during reduction. In contrast to FJ,
 313 where **new** C() is a value, in **FHJ** such an expression is not a value. Instead, an expression
 314 of the form **new** C() is reduced to (C)new C(), which is a value in **FHJ** and the cast denotes

the static type of the expression. This rule is applied in the first reduction step. In the second reduction step, when `shuffleAndDraw` is dispatched, the receiver (`DrawableSafeDeck`) `new DrawableSafeDeck()` replaces the special variable `this` by `(Deck)new DrawableSafeDeck()`. Here, the static type used in the cast `(Deck)` denotes the origin of the `shuffleAndDraw` method, which is discovered during method lookup. Later, in the fourth step, `((Deck)new DrawableSafeDeck()).draw()` is an instance of *hierarchical invocation*, which can be read as “finding the most specific `draw` above `DrawableSafeDeck` and along path `Deck`”. The meaning of “above `DrawableSafeDeck`” implies its supertypes, and “along path `Deck`” specifies the branch. Finally, in the last reduction step, we find the most specific version of `draw` in `SafeDeck`. In this sequence of reduction steps, the cast that tracks the origin of `shuffleAndDraw` is crucial to unambiguously find the correct implementation of `draw`. The formal procedure will be introduced in Section 3 and Section 4.

2.3 Problem 3: Overriding on Individual Branches

Method overriding is common in Object-Oriented Programming. With diamond inheritance, where conflicting methods are intended to have the same semantics, method overriding is not a problem. If conflicting methods arise from multiple parents, we can override all those methods in a single unified (or merged) method in the subclass. Therefore further overriding is simple, because there is only one method that can be overridden.

With unintentional method conflicts, however, the situation is more complicated because different, separate, conflicting methods can coexist in one class. Ideally, we would like to support overriding for those methods too, in exactly the same way that overriding is available for other (non-conflicting) methods. However, we need to be able to override the individual conflicting methods, rather than overriding all conflicting methods into a single merged one.

We illustrate the problem and the need for a more refined overriding mechanism with an example. Suppose that the programmer defines a new interface `DrawableSafeDeck` (based on the code in Section 2.2 without the old `DrawableSafeDeck`), but he needs to override `Drawable.draw` and give a new implementation of drawing so that the deck can indeed be visualized on the canvas.

2.3.1 Potential solutions/workarounds in existing languages

YANLIN: Bruno: one reviewer says S2.3.1 is "less clear", could you check again?BRUNO: seems ok to me. Unfortunately in all languages we know of (including C++), the existing approaches are unsatisfactory. One direction is to simply avoid this issue, by putting overriding before inheritance. For example, as shown in Figure 2(middle), we define a new component `DrawableRect` that extends `Drawable`, which simply draws the deck as a rectangle, and modifies the hierarchy:

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

This workaround seems to work, but there are severe issues:

- It changes the hierarchy and existing code, hence breaks the modularity.

363 ■ Separate overriding is required to come after the fork inheritance, especially when the
 364 implementation needs functionality from both parents. In the above code, we have
 365 assumed that the overriding is unrelated to `Deck`. But when the drawing relies on some
 366 information of the `Deck` object, we have to either introduce field methods for delegation
 367 or change the signature of `draw` to take a parameter. Either way introduces unnecessary
 368 complexity and affects extensibility.

369 There are more involved workarounds in C++ using templates and complex patterns,
 370 but such patterns are complex to use and there are still issues. A more detailed discussion of
 371 such an approach is presented in Section 6.2.

372 2.3.2 FHJ's solution

373 An additional feature of our model is *hierarchical overriding*. It allows conflicting methods
 374 to be overridden on individual branches, hence offers independent extensibility. The above
 375 example can be easily realized by:

```
376 interface DrawableSafeDeck extends Drawable, SafeDeck {
377     void draw() override {
378         JFrame frame = new JFrame("Canvas");
379         frame.setSize(600, 600);
380         frame.getContentPane().setBackground(Color.red);
381         frame.getContentPane().add(new Square(10,10,100,100)); ...
382     }
383 }
384 // main program
385 ((Drawable)new DrawableSafeDeck()).draw(); //calls the draw in DrawableSafeDeck
386
387
```

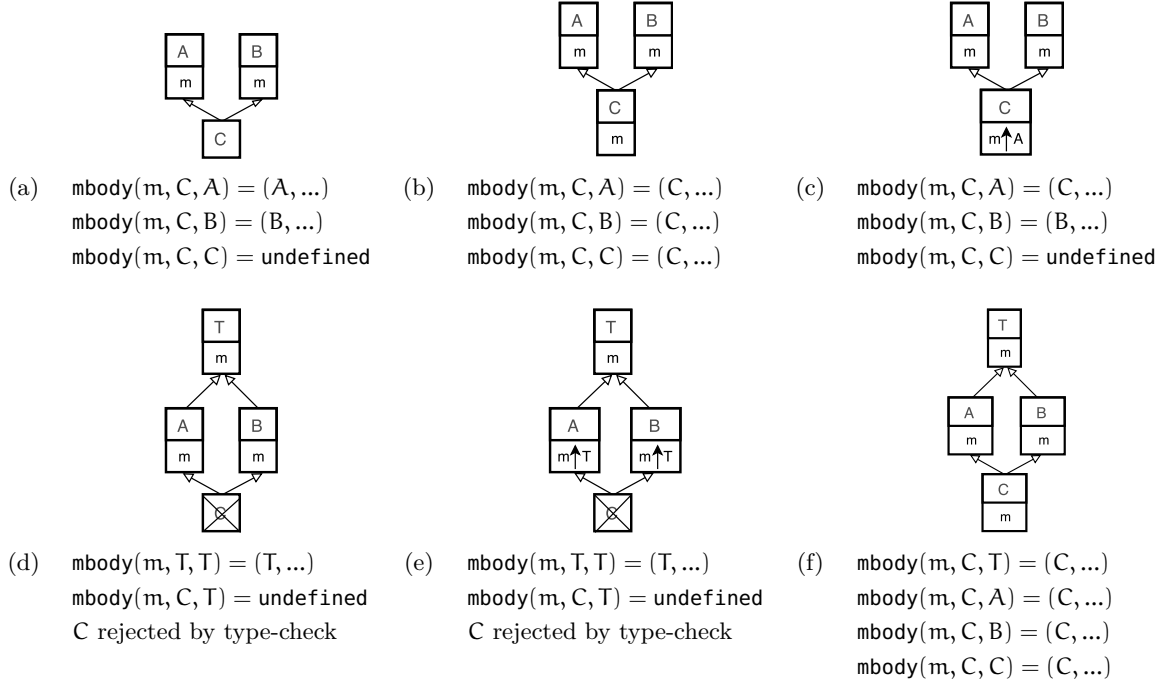
388 The UML graph is shown in Figure 2(right), where the up-arrow ↑ is short for “**override**”.
 389 Here the idea is that *only* `Drawable.draw` is overridden. This is accomplished by specifying, in
 390 the method definition, that the method only overrides the `draw` from `Drawable`. The individual
 391 overriding allows us to make use of the methods from `SafeDeck` as well. In the formalization,
 392 the hierarchical overriding feature is an important feature, involved in the algorithm of
 393 hierarchical dispatch.

394 Note that, although the example here only shows one conflicting method being overridden,
 395 hierarchical overriding allows (as expected) multiple conflicting methods to be overridden in
 396 the same class.

397 2.3.3 Terminology

398 In `Drawable`, `Deck`, and `SafeDeck`, the `draw` methods are called *original methods* in this paper,
 399 because they are originally defined by the interfaces. In contrast, `DrawableSafeDeck` defines a
 400 *hierarchical overriding method*. The difference is that traditional method overriding overrides
 401 all branches by defining another original method, whereas hierarchical overriding only refines
 402 one branch.

403 A special rule for hierarchical overriding is: it can only refine *original* methods, and cannot
 404 jump over original methods with the same signature. For instance, writing “`void draw()`
 405 **override** `Deck {...}`” is disallowed in `DrawableSafeDeck`, because the existing two branches
 406 are `Drawable.draw` and `SafeDeck.draw`, while `Deck.draw` is already covered. It does not really
 407 make sense to refine the old branch `Deck`.



■ **Figure 3** Examples in **FHJ**. “ $m \uparrow A$ ” stands for hierarchical overriding “ m override A ”.

2.3.4 A peek at the hierarchical dispatching algorithm

In **FHJ**, fork inheritance allows several original methods (branches) to coexist, and hierarchical dispatch first finds the most specific original method (branch), then it finds the most specific hierarchical overriding on that branch.

Before the formalized algorithm, Figure 3 gives a peek at the behavior using a few examples. The UML diagrams present the hierarchy. In (d) and (e), a cross mark indicates that the interface fails to type-check. Generally, **FHJ** rejects the definition of an interface during compilation if it reaches a diamond with ambiguity. **mbody** is the method lookup function for hierarchical dispatch, formally defined in Section 4.1. In general, $\text{mbody}(m, X, Y) = (Z, \dots)$ reflects that the source code $((Y)\text{new } X()).m()$ calls $Z.m$ at runtime. It is undefined when method dispatch is ambiguous.

In Figure 3, (a) is the base case for unintentional conflicts, namely the fork inheritance. (b) uses overriding to explicitly merge the conflicting methods. (c) represents hierarchical overriding. Our model supports diamond inheritance and can deal with diamond problems. For example, (d) and (e) are two base cases of diamond inheritance in **FHJ** and the definition of each C is rejected because T is an ambiguous parent to C . One solution for diamond inheritance is to merge methods coming from different parents. (f) gives a common solution to the diamond as in Java or traits, which is to explicitly override $A.m$ and $B.m$ in C . And our calculus supports this kind of merging methods. In the last three examples, conflicting methods $A.m$ and $B.m$ should be viewed as intentional conflicts, as they come from the same source T .

BRUNO: I remember that a big complain in the reviews was talking about diamond inheritance. The Figure does have diamond inheritance but this wasn't clear to the reviewers. Please expand the text to emphasize diamond inheritance better! YANLIN: expanded.

3 Formalization

In this section, we present a formal model called **FHJ** (*Featherweight Hierarchical Java*), following a similar style as Featherweight Java [14]. **FHJ** is a minimal core calculus that formalizes the core concept of hierarchical dispatching and overriding. The syntax, typing rules and small-step semantics are presented.

3.1 Syntax

The abstract syntax of **FHJ** interface declarations, method declarations, and expressions is given in Figure 4. The multiple inheritance feature of **FHJ** is inspired by Java 8 interfaces, which supports method implementations via default methods. This feature is closely related to *traits*. To demonstrate how unintentional method conflicts are untangled in **FHJ**, we only focus on a small subset of the interface model. For example, all methods declared in an interface are either default methods or abstract methods. Default methods provide default implementations for methods. Abstract methods do not have a method body. Abstract methods can be overridden with future implementations.

3.1.1 Notations

The metavariables I, J, K range over interface names; x ranges over variables; m ranges over method names; e ranges over expressions; and M ranges over method declarations. Following Featherweight Java, we assume that the set of variables includes the special variable **this**, which cannot be used as the name of an argument to a method. We use the same conventions as FJ; we write \bar{I} as shorthand for a possibly empty sequence I_1, \dots, I_n , which may be indexed by I_i ; and write \bar{M} as shorthand for $M_1 \dots M_n$ (with no commas). We also abbreviate operations on pairs of sequences in an obvious way, writing $\bar{I} \bar{x}$ for $I_1 x_1, \dots, I_n x_n$, where n is the length of \bar{I} and \bar{x} .

3.1.2 Interfaces

YANLIN: Bruno: acyclic, clarified herer. In order to achieve multiple inheritance, an interface can have a set of parent interfaces, where such a set can be empty. Moreover, as usual in class-based languages, **YANLIN: refined 'acyclic'** the extension relation over interfaces is acyclic. The interface declaration **interface** I **extends** $\bar{I} \{ \bar{M} \}$ introduces an interface named I with parent interfaces \bar{I} and a suite of methods \bar{M} . The methods of I may either override methods that are already defined in \bar{I} or add new functionality special to I , we will illustrate this in more detail later.

3.1.3 Methods

Original methods and hierarchically overriding methods share the same syntax in our model for simplicity. The concrete method declaration $I \ m(\bar{I}_x \ \bar{x}) \ \text{override} \ J \ \{ \text{return } e; \}$ introduces a method named m with result type I , parameters \bar{x} of type \bar{I}_x and the overriding target J . The body of the method simply includes the returned expression e . Notably, we have introduced the **override** keyword for two cases: if the overridden interface is exactly the enclosing interface itself, then such a method is seen as originally defined **YANLIN: added an explanation** (including the case of merging methods from different branches, which can be viewed as *reborn*); otherwise it is a hierarchical overriding method. Note that in an interface J , $I \ m(\bar{I}_x \ \bar{x}) \ \{ \text{return } e; \}$ is syntactic sugar for $I \ m(\bar{I}_x \ \bar{x}) \ \text{override} \ J \ \{ \text{return } e; \}$, which is the

Interfaces	IL	$::=$	<code>interface I extends \bar{I} {\bar{M}}</code>
Methods	M	$::=$	<code>I $m(\bar{I}_x \bar{x})$ override J {return e; } I $m(\bar{I}_x \bar{x})$ override J ;</code>
Expressions	e	$::=$	<code>x $e.m(\bar{e})$ <code>new $I()$</code> $(I)e$</code>
Context	Γ	$::=$	<code>$\bar{x} : \bar{I}$</code>
Values	v	$::=$	<code>(I)new $J()$</code>

■ **Figure 4** Syntax of **FHJ**.

standard way of method definition in Java-like languages. The definition of abstract methods is written as `I $m(\bar{I}_x \bar{x})$ override J ;`, which is similar to a concrete method but without the method body. For simplicity, overloading is not modelled for methods, which implies that we can uniquely identify a method by its name.

3.1.4 Expressions & Values

Expressions can be standard constructs such as variables, method invocation, object creation, together with cast expressions. Object creation is represented by `new $I()$` ². Fields and primitive types are not modelled in **FHJ**. The casts are merely safe upcasts, and in fact, they can be viewed as annotated expressions, where the annotation indicates its static type. The coexistence of static and dynamic types is the key to hierarchical dispatch. A value “ (I) new $J()$ ” is the final result of multiple reduction steps evaluating an expression.

For simplicity, **FHJ** does not formalize statements like assignments and so on because they are orthogonal features to the hierarchical dispatching and overriding feature. A program in **FHJ** consists of a list of interface declarations, plus a single expression.

3.2 Subtyping and Typing Rules

3.2.1 Subtyping

The subtyping of **FHJ** consists of only a few rules shown at the top of Figure 5. In short, subtyping relations are built from the inheritance in interface declarations. Subtyping is both reflexive and transitive.

3.2.2 Type-checking

Details of type-checking rules are displayed at the bottom of Figure 5, including expression typing, well-formedness of methods and interfaces. As a convention, an environment Γ is maintained to store the types of variables, together with the self-reference `this`.

(T-INVK) is the typing rule for method invocation. Naturally, the receiver and the arguments are required to be well-typed. `mbody` is our key function for method lookup that implements the hierarchical dispatching algorithm. The formal definition will be introduced in Section 4. Here `mbody(m, I_0, I_0)` finds the most specific m above I_0 . “Above I_0 ” specifies the search space, namely the supertypes of I_0 including itself. For the general case, however, the hierarchical invocation `mbody(m, I, J)` finds “the most specific m above I and along path/branch J ”. “Along path J ” additionally requires the result to relate to J , that is to say, the most specific interface that has a subtyping relationship with J .

² In Java the corresponding syntax is `new $I()$ {}`.

$$\begin{array}{c}
\boxed{I <: J} \qquad \overline{I <: I} \\
\\
\frac{I <: J \quad J <: K}{I <: K} \qquad \frac{\text{interface } I \text{ extends } I_1, I_2, \dots, I_n \{\overline{M}\}}{I <: I_1, I <: I_2, \dots, I <: I_n} \\
\\
\boxed{\Gamma \vdash e : I} \qquad (\text{T-VAR}) \Gamma \vdash x : \Gamma(x) \\
\\
(\text{T-INVK}) \frac{\Gamma \vdash e_0 : I_0 \quad \text{mbody}(m, I_0, I_0) = (K, \bar{J} \bar{x}, I _) \quad \Gamma \vdash \bar{e} : \bar{I} \quad \bar{I} <: \bar{J}}{\Gamma \vdash e_0.m(\bar{e}) : I} \\
\\
(\text{T-NEW}) \frac{\text{interface } I \text{ extends } \bar{I} \{\overline{M}\} \quad \text{canInstantiate}(I)}{\Gamma \vdash \text{new } I() : I} \\
\\
(\text{T-ANNO}) \frac{\Gamma \vdash e : I \quad I <: J}{\Gamma \vdash (J)e : J} \\
\\
(\text{T-METHOD}) \frac{I <: J \quad \text{findOrigin}(m, I, J) = \{J\} \quad \text{mbody}(m, J, J) = (K, \bar{I}_x \bar{x}, I_e _) \quad \bar{x} : \bar{I}_x, \text{this} : I \vdash e_0 : I_0 \quad I_0 <: I_e}{I_e \ m(\bar{I}_x \bar{x}) \text{ override } J \{ \text{return } e_0; \} \text{ OK IN } I} \\
\\
(\text{T-ABSMETHOD}) \frac{I <: J \quad \text{findOrigin}(m, I, J) = \{J\} \quad \text{mbody}(m, J, J) = (K, \bar{I}_x \bar{x}, I_e _)}{I_e \ m(\bar{I}_x \bar{x}) \text{ override } J ; \text{ OK IN } I} \\
\\
\overline{M} \text{ OK IN } I \\
\forall J :> I \text{ and } m, \text{mbody}(m, J, J) \text{ is defined} \Rightarrow \text{mbody}(m, I, J) \text{ is defined} \\
(\text{T-INTF}) \frac{\forall J :> I \text{ and } m, I[m \text{ override } I] \text{ and } J[m \text{ override } J] \text{ defined} \Rightarrow \text{canOverride}(m, I, J)}{\text{interface } I \text{ extends } \bar{I} \{\overline{M}\} \text{ OK}}
\end{array}$$

■ **Figure 5** Subtyping and Typing Rules of **FHJ**.

504 In (T-INVK), as the compilation should not be aware of the dynamic type, it only requires
 505 that invoking `m` is valid for the static type of the receiver. The result of `mbody` contains
 506 the interface that provides the most specific implementation, the parameters and the return
 507 type. We use underscore for the return expression, matching both implemented and abstract
 508 methods. **YANLIN: refined.**

509 (T-NEW) is the typing rule for object creation `new I()`. The auxiliary function `canInstantiate(I)`
 510 (see definition in Section 4.4) checks whether an interface `I` can be instantiated or not. Since
 511 fork inheritance accepts conflicting branches to coexist, the check requires that the most
 512 specific method is concrete for each method on each branch.

513 (T-METHOD) is more interesting since a method can either be an original method
 514 or a hierarchical overriding, though they share the same syntax and method typing rule.
 515 `findOrigin(m, I, J)` is a fundamental function, used to find “the most specific interfaces that
 516 are above `I` and along path `J`, and originally defines `m`” (see Section 4 for full definition).
 517 By “most specific interfaces”, it implies that the inherited supertypes are excluded. Thus

the condition $\text{findOrigin}(m, I, J) = \{J\}$ indicates a characteristic of a hierarchical overriding: it must override an original method; the overriding is direct and there does not exist any other original method m in between. Then $\text{mbody}(m, J, J)$ provides the type of the original method, so hierarchical overriding has to preserve the type. Finally the return expression is type-checked to be subtype of the declared return type. For the definition of an original method, I equals J and the rule is straightforward. (T-ABSMETHOD) is a similar rule but works on abstract method declarations.

(T-INTF) defines the typing rule on interfaces. The first condition is obvious, namely its methods need to be well checked. The third condition checks whether the overriding between original methods preserves typing. In this condition we again use some helper functions defined in Section 4. $I[m \text{ override } I]$ is defined if I originally defines m , and $\text{canOverride}(m, I, J)$ checks whether $I.m$ has the same type as $J.m$. Generally the preservation of method type is required for any supertype J and any method m .

The second condition of (T-INTF) is more complex and is the key to type soundness. Unlike C++ which rejects on ambiguous calls, **FHJ** rejects on the definition of interfaces when they form a diamond. Consider the case when the second condition is broken: $\text{mbody}(m, J, J)$ is defined but $\text{mbody}(m, I, J)$ is undefined for some J and m . This indicates that m is available and unambiguous from the perspective of J , but is ambiguous to I on branch J . It means that there are multiple overriding paths of m from J to I , which form a diamond. Hence rejecting that case meets our expectation. Below is an example (Figure 3 (e)) that illustrates the reason why this condition is needed:

```

540 interface T { T m() override T { return new T(); } }
541 interface A extends T { T m() override T { return new A(); } }
542 interface B extends T { T m() override T { return new B(); } }
543 interface C extends A, B {}
544 ((T) new C()).m()

```

This program does not compile on interface C , because of the second condition in (T-INTF), where I equals C and J equals T . By the algorithm, $\text{mbody}(m, T, T)$ will refer to $T.m$, but $\text{mbody}(m, C, T)$ is undefined, since both $A.m$ and $B.m$ are most specific to C along path T , which forms a diamond. The expression $((T)\text{new } C()).m()$ is one example of triggering ambiguity, but **FHJ** simply rejects the definition of C . To resolve the issue, the programmer needs to have an overriding method in C , to explicitly merge the conflicting ones.

Finally, rule (T-ANNO) is the typing rule for a cast expression. By the rule, only upcasts are valid.

3.3 Small-step Semantics and Propagation

Figure 6 defines the small-step semantics and propagation rules of **FHJ**. When evaluating an expression, they are invoked and produce a single value in the end.

3.3.1 Semantic Rules

(S-INVK) is the only computation rule we need for method invocation. As a small-step rule and by congruence, it assumes that the receiver and the arguments are already values. Specifically, the receiver $(J)\text{new } I()$ indicates the dynamic type I together with the static type J . Therefore $\text{mbody}(m, I, J)$ carries out hierarchical dispatching, acquires the types, the return expression e_0 and the interface I_0 which provides the most specific method. Here we use e_0 to imply that the return expression is forced to be non-empty because it requires a concrete implementation. Now the rule reduces method invocation to e_0 with substitution.

Parameters are substituted with arguments, and the **this** reference is substituted with the receiver, and in the meanwhile the static types are recorded via annotations. Finally, the return type I_e is put in the front as an annotation.

3.3.2 Propagation Rules

(C-RECEIVER), (C-ARGS) and (C-FREDUCE) are natural propagation rules on receivers, arguments, and cast-expressions, respectively. (C-STATIC TYPE) automatically adds an annotation I to the new object **new** $I()$. (C-ANNOREDUCE) merges nested upcasts into a single upcast with the outermost type.

4 Key Algorithms and Type-Soundness

In this section, we present the fundamental algorithms and auxiliary definitions used in our formalization and show that the resulting calculus is type sound. The functions presented in this section are the key components that implement our algorithm for method lookup.

4.1 The Method Lookup Algorithm in **mbody**

mbody(m, I_d, I_s) denotes the method body lookup function. We use I_d, I_s , since **mbody** is usually invoked by a receiver of a method m , with its dynamic type I_d and static type I_s . Such a function returns the most specific method implementation. More accurately, **mbody** returns the parameters, returned expression (empty for abstract methods) and the types for the method. It considers both originally-defined methods and hierarchical overriding methods, so **findOrigin** and **findOverride** (see the definition in Section 4.2 and Section 4.3) are both invoked. The formal definition gives the expected results for the earlier examples in Figure 3.

▷ *Definition of mbody*(m, I_d, I_s) :

- **mbody**(m, I_d, I_s) = ($J, \overline{I_x} \overline{x}, I_e e_0$)
 with: **findOrigin**(m, I_d, I_s) = $\{I\}$
findOverride(m, I_d, I) = $\{J\}$
 $J[m \text{ override } I] = I_e m(\overline{I_x} \overline{x}) \text{ override } I \{ \text{return } e_0; \}$
- **mbody**(m, I_d, I_s) = ($J, \overline{I_x} \overline{x}, I_e \emptyset$)
 with: **findOrigin**(m, I_d, I_s) = $\{I\}$
findOverride(m, I_d, I) = $\{J\}$
 $J[m \text{ override } I] = I_e m(\overline{I_x} \overline{x}) \text{ override } I ;$

To calculate **mbody**(m, I_d, I_s), the invocation of **findOrigin** looks for the most specific original methods and their interfaces, and expects a singleton set, so as to achieve unambiguity. Furthermore, the invocation of **findOverride** also expects a unique and most specific hierarchical override. And finally the target method is returned.

$$\begin{array}{c}
\text{(S-INVK)} \frac{\text{mbody}(\mathbf{m}, \mathbf{I}, \mathbf{J}) = (\mathbf{I}_0, \overline{\mathbf{I}_x} \ \overline{x}, \mathbf{I}_e \ e_0)}{((\mathbf{J})\text{new } \mathbf{I}()) . \mathbf{m}(\overline{v}) \rightarrow (\mathbf{I}_e)[\overline{(\mathbf{I}_x)}\overline{v}/\overline{x}, (\mathbf{I}_0)\text{new } \mathbf{I}()/\text{this}]e_0} \\
\\
\text{(C-RECEIVER)} \frac{e_0 \rightarrow e'_0}{e_0 . \mathbf{m}(\overline{e}) \rightarrow e'_0 . \mathbf{m}(\overline{e})} \quad \text{(C-ARGS)} \frac{e \rightarrow e'}{e_0 . \mathbf{m}(\dots, e, \dots) \rightarrow e_0 . \mathbf{m}(\dots, e', \dots)} \\
\\
\text{(C-STATIC TYPE)} \frac{}{\text{new } \mathbf{I}() \rightarrow (\mathbf{I})\text{new } \mathbf{I}()} \\
\\
\text{(C-FREDUCE)} \frac{e \rightarrow e' \quad e \neq \text{new } \mathbf{J}()}{(\mathbf{I})e \rightarrow (\mathbf{I})e'} \\
\\
\text{(C-ANNOREDUCE)} (\mathbf{I})((\mathbf{J})\text{new } \mathbf{K}()) \rightarrow (\mathbf{I})\text{new } \mathbf{K}()
\end{array}$$

■ **Figure 6** Small-step semantics.

602 4.2 Finding the Most Specific Origin: `findOrigin`

603 We proceed to give the definitions of two core functions that support method lookup, namely
 604 `findOrigin` and `findOverride`. Generally, `findOrigin(m, I, J)` finds the set of most specific
 605 interfaces where `m` is originally defined. Interfaces in this set should be above interface `I`
 606 and along path `J`. Finally with `prune` (defined in Section 4.4) the overridden interfaces will
 607 be filtered out.

608 ▷ *Definition of `findOrigin(m, I, J)`:*
 609 • `findOrigin(m, I, J) = prune(origins)`
 610 with: `origins = {K | I <: K, and K <: J ∨ J <: K,`
 611 `and K[m override K] is defined}`
 612

613 By the definition, an interface belongs to `findOrigin(m, I, J)` if and only if:

- 614 ■ It originally defines `m`;
- 615 ■ It is a supertype of `I` (including `I`);
- 616 ■ It is either a supertype or a subtype of `J` (including `J`);
- 617 ■ Any subtype of it does not belong to the same result set because of `prune`.

618 4.3 Finding the Most Specific Overriding: `findOverride`

619 The `findOrigin` function only focuses on original method implementations, where all
 620 the hierarchical overriding methods are omitted during that step. On the other hand,
 621 `findOverride(m, I, J)` has the assumption that `J` defines an original `m`, and this function
 622 tries to find the interfaces with the most specific implementations that hierarchically overrides
 623 such an `m`. Formally,

624 \triangleright *Definition of findOverride(m, I, J) :*
 625 • $\text{findOverride}(m, I, J) = \text{prune}(\text{overrides})$
 626 with: $\text{overrides} = \{K \mid I <: K, K <: J \text{ and } K[m \text{ override } J] \text{ is defined}\}$
 627

628
 629 By the definition, an interface belongs to $\text{findOverride}(m, I, J)$ if and only if:

- 630 ■ it is between I and J (including I, J);
- 631 ■ it hierarchically overrides J.m;
- 632 ■ any subtype of it does not belong to the same set.

633 4.4 Other Auxiliaries

634 Below we give other minor definitions of the auxiliary functions that are used in previous
 635 sections.

636 \triangleright *Definition of I[m override J] :*
 637 • $I[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \{ \text{return } e_0; \}$
 638 with: interface I extends $\overline{I} \{ I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \{ \text{return } e_0; \} \dots \}$
 639 • $I[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J ;$
 640 with: interface I extends $\overline{I} \{ I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J ; \dots \}$
 641
 642

643 Here $I[m \text{ override } J]$ is basically a direct lookup for method m in the body of I , where such
 644 a method overrides J (like static dispatch). The method can be either concrete or abstract,
 645 and the body of definition is returned. Notice that by our syntax, $I[m \text{ override } I]$ is looking
 646 for the originally-defined method m in I .

647 \triangleright *Definition of prune(set) :*
 648 • $\text{prune}(\text{set}) = \{I \in \text{set} \mid \nexists J \in \text{set} \setminus I, J <: I\}$
 649

650 The prune function takes a set of types, and filters out those that have subtypes in the
 651 same set. In the returned set, none of them has subtyping relation to one another, since all
 652 supertypes have been removed.

653 \triangleright *Definition of canOverride(m, I, J) :*
 654 • $\text{canOverride}(m, I, J) = \text{True}$
 655 with: $I[m \text{ override } I] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } I \dots$
 656 $J[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{y}) \text{ override } J \dots$
 657

658 canOverride just checks that two original m in I and J have the same type.

659 \triangleright *Definition of* $\text{canInstantiate}(I)$:

660 • $\text{canInstantiate}(I) = \text{True}$

661 with: $\forall m, \forall J \in \text{findOrigin}(m, I, I), \text{findOverride}(m, I, J) = \{K\},$

662 and $K[m \text{ override } J] = I_e \ m(\overline{I_x} \ \overline{x}) \text{ override } J \{\text{return } e_0;\}$

663

664 $\text{canInstantiate}(I)$ checks whether interface I can be instantiated by the keyword **new**.

665 $\text{findOrigin}(m, I, I)$ represents the set of branches I inherits on method m . I can be instanti-

666 ated if and only if for every branch, the most specific implementation is unambiguous and

667 non-abstract.

668 4.5 Properties

669 We present the type soundness of the model by a few theorems below, following the standard

670 technique of subject reduction and progress proposed by Wright and Felleisen [35]. The

671 proof, together with some lemmas, is presented in Appendix. Type soundness states that if

672 an expression is well-typed, then after many reduction steps it must reduce to a value, and

673 its annotation is the same as the static type of the original expression.

674 **► Theorem 1** (Subject Reduction). *If $\Gamma \vdash e : I$ and $e \rightarrow e'$, then $\Gamma \vdash e' : I$.*

675 **Proof.** See Appendix A.1. ◀

676 **► Theorem 2** (Progress). *Suppose e is a well-typed expression, if e includes $((J)\text{new } I()) \cdot m(\overline{v})$*

677 *as a sub-expression, then $\text{mbody}(m, I, J) = (I_0, \overline{I_x} \ \overline{x}, I_e \ e_0)$ and $\#(\overline{x}) = \#(\overline{v})$ for some $I_0, \overline{I_x},$*

678 *\overline{x}, I_e and e_0 .*

679 **Proof.** See Appendix A.1. ◀

680 **► Theorem 3** (Type Soundness). *If $\phi \vdash e : I$ and $e \rightarrow^* e'$ with e' a normal form, then e' is*

681 *a value v with $\phi \vdash v : I$.*

682 **Proof.** Immediate from Theorem 1 and Theorem 2. ◀

683 Note that in Theorem 2, “ $\#(\overline{x})$ ” denotes the length of \overline{x} .

684 Our theorems are stricter than those of Featherweight Java [14]. In FJ, the subject

685 reduction theorem states that after a step of reduction, the type of an expression may change

686 to a subtype due to subtyping. However, in **FHH**, the type remains unchanged because we

687 keep track of the static types and use them for casting during reduction.

688 Finally we show that one-step evaluation is deterministic. This theorem is helpful to

689 show that our model of multiple inheritance is not ambiguous (or non-deterministic).

690 **► Theorem 4** (Determinacy of One-Step Evaluation). *If $t \rightarrow t'$ and $t \rightarrow t''$, then $t' = t''$.*

691 **Proof.** See Appendix A.1. ◀

692 5 Discussion

693 In this section, we will discuss the design space and reflect about some of the design decisions

694 of our work. We relate our language to traits, Java interfaces as well as other languages.

695 Furthermore, we discuss ways to improve our work.



■ **Figure 7** Fork inheritance (left) and diamond inheritance (right) on abstract methods.

5.1 Abstract Methods

Abstract methods are one of the key features in most general OO languages. For example, Java interfaces were designed to include only method declarations, and those abstract methods can be implemented in a class body. The formal Featherweight Java model [14] does not include abstract methods because of the orthogonality to the core calculus. In traits, a similar idea is to use keywords like “**require**” for abstract method declarations [28]. Abstract methods provide a way to delay the implementations to future subtypes. Using overriding, they also help to “exclude” existing implementations.

In our formalized calculus, however, abstract methods are not a completely orthogonal feature. The `canInstantiate` function has to check whether an interface can be instantiated by looking at all the inherited branches and checking if each most specific method is concrete or not.

Our formalization has a simple form of abstract methods, which behave similarly to conventional methods with respect to conflicts. Other languages may behave differently. For instance, in Java 8 when putting two identical abstract methods together by multiple inheritance, there is no conflict error. In Figure 7, we use italic *m* to denote abstract methods. In both cases, the Java compiler accepts the definition of C and automatically merges the two inherited methods *m* into a single one. **FHJ** behaves differently from Java in both cases. In the fork inheritance case (left), C will have two distinct abstract methods corresponding to *A.m* and *B.m*. In the diamond inheritance case, the definition of C is rejected. There are two reasons for this difference in behaviour. Firstly, our formalization just treats abstract methods as concrete methods with an empty body, and that simplifies the rules and proofs a lot. Secondly, and more importantly, we distinguish and treat differently conflicting methods, since they may represent different operations, even if they are abstract. Thus our model adopts a very conservative behavior rather than automatically merging methods by default (as done in many languages). Arguably, for the diamond case, as we have mentioned, it is actually an intentional conflict due to the same source T. It is possible to change our model to account for other behaviors for abstract methods, but we view this as a mostly orthogonal change to our work, which should not affect the essence of the model presented here.

5.2 Orthogonal & Non-Orthogonal Extensions

Our model is designed as a minimal calculus that focuses on resolving unintentional conflicts. Therefore, we have omitted a number of common orthogonal features including primitive types, assignments, method overloading, covariant method return types, static dispatch, and so on. Those features can, in principle, be modularly added to the model without breaking type soundness. For example, we present the additional syntax, typing and semantic rules of static invocation below as an extension:

Expressions $e ::= \dots \mid e.J_0@J_1 :: m(\bar{e})$

$$(T\text{-STATICINVK}) \frac{J_0[m \text{ override } J_1] = I \ m(\bar{J} \ \bar{x}) \text{ override } J_1 \ \{\text{return } e;\} \quad \Gamma \vdash e_0 : I_0 \quad I_0 <: J_0 \quad \Gamma \vdash \bar{e} : \bar{I} \quad \bar{I} <: \bar{J}}{\Gamma \vdash e_0.J_0@J_1 :: m(\bar{e}) : I}$$

$$(S\text{-STATICINVK}) \frac{J_0[m \text{ override } J_1] = I_e \ m(\bar{I}_x \ \bar{x}) \text{ override } J_1 \ \{\text{return } e_0;\} \quad ((J)\text{new } I()) . J_0@J_1 :: m(\bar{v}) \rightarrow (I_e)[(\bar{I}_x)\bar{v}/\bar{x}, (J_0)\text{new } I()/\text{this}]e_0}{}$$

A static invocation $e.J_0@J_1 :: m(\bar{e})$ aims at finding the method m in J_0 that hierarchically overrides J_1 , thus $J_0[m \text{ override } J_1]$ is invoked. As shown in (S-STATICINVK), static dispatch needs a receiver for the substitution of the “this” reference, so as to provide the latest implementations. In fact, static dispatch is common in OO programming, as it provides a shortcut to the reuse of old implementation easily, and super calls can also rely on this feature. For convenience we just make it simple above, whereas in languages like C++ or Java, the static or super invocations are more flexible, as they can climb the class hierarchy.

YANLIN: Bruno: discussed multiple overriding and the two designs One non-orthogonal extension to **FHJ** could be to generalize the model to allow multiple hierarchical method overriding, meaning that, we allow overriding methods to update multiple branches instead of only one branch. It offers a more fine-grained mechanism for merging, and can be helpful to easily understand the structure of the hierarchy. Multiple overriding would be useful in the following situation, for example:

```

745 interface A { void m() {...} }
746 interface B { void m() {...} }
747 interface C { void m() {...} }
748 interface D extends A, B, C {
749   void m() override A,B {...} // overrides branches A and B only
750   void m() override C {...} // overrides branch C
751 }
752
753
```

Here **D** inherits from three interfaces **A**, **B**, **C** with conflicting methods m , but only merges two of those methods. While we can simulate **D** without multiple overriding in our calculus (by introducing an intermediate class), a better approach would be to support multiple overriding natively.

We present the modification of syntax, typing and semantic rules below (abstract methods omitted):

Methods $M ::= \dots \mid I \ m(\bar{I}_x \ \bar{x}) \text{ override } \bar{J} \ \{\text{return } e;\}$

$$(T\text{-MOMETHOD}) \frac{\forall J_i \in \bar{J}, I <: J_i \quad \text{findOrigin}(m, I, J_i) = \{J_i\} \quad \text{mbody}(m, J_i, J_i) = (K, \bar{I}_x \ \bar{x}, I_e \ _) \quad \bar{x} : \bar{I}_x, \text{this} : I \vdash e_0 : I_0 \quad I_0 <: I_e}{I_e \ m(\bar{I}_x \ \bar{x}) \text{ override } \bar{J} \ \{\text{return } e_0;\} \text{ OK IN } I}$$

Semantic rules themselves remain unchanged, however, we need to change slightly the definition of `findOverride` in `mbody`:

762 \triangleright *Definition of* `findOverride(m, I, J)` :
 763 • `findOverride(m, I, J) = prune(overrides)`
 764 with: `overrides = {K | I <: K, K <: J and K[m override J]}` where $J \in \bar{J}$
 765

766
 767 With this approach, branches **A** and **B** are merged in the sense that they share the same
 768 code, which can be separately updated in future interfaces. Another approach is to deeply
 769 merge the branches, with similar effect as introducing an intermediate interface **AB** to explicitly
 770 merge the two branches. However, this approach is problematic because there is no clear
 771 mechanism of identifying and further updating the merged branches. This could be an
 772 interesting future work to explore.

773 Other typical non-orthogonal extensions to **FHJ** could be to have fields. The design of
 774 **FHJ** can be viewed as a variant of Java 8 with default methods which allows for unintentional
 775 method conflicts. Like Java interfaces and traits, state is forbidden in **FHJ**. There are some
 776 inheritance models that also account for fields, such as C++ that uses virtual inheritance [10].
 777 In our model, however, we can perhaps borrow the idea of *interface-based programming* [33],
 778 which models state with abstract state operations. This can be realized by extending our
 779 current model with static methods and anonymous classes from Java. However such an
 780 extension requires more thought, so we leave it to future work.

781 5.3 Loosening the Model: Reject Early or Reject Later?

782 **FHJ** rejects the following case of diamond inheritance:

```
783 interface A { void m() {...} }
784 interface B extends A { void m() {...} }
785 interface C extends A { void m() {...} }
786 interface D extends B, C {}
787
```

789 Here both `B.m` and `C.m` override `A.m`, and `D` inherits both conflicting methods without an
 790 explicit override. In this case, automatically merging the two methods (to achieve diamond
 791 inheritance) is not possible, which is why many models (like traits and Java 8) reject such
 792 programs. Moreover, keeping the two method implementations in `D` is problematic. In essence,
 793 hierarchical information is not helpful to disambiguate later method calls, since the two
 794 methods share the same origin (`A.m`). Our calculus rejects such conflicts by the (T-INTF)
 795 rule, where `D` is considered to be ill-formed. We believe that rejecting `D` follows the principle
 796 of models like traits and Java 8 interfaces, where the language/type-system is meant to alert
 797 the programmer for a possible conflict early.

798 Nonetheless, C++ accepts the definition of `D`, but forbids later upcasts from `D` to `A` because
 799 of ambiguity. Our language is more conservative on definitions of interfaces compared to
 800 C++, but on the upside, upcasts are not rejected. We could also loosen the model to accept
 801 definitions such as `D`, and perform ambiguity check on upcasts and other expressions. Then,
 802 we would need to handle more cases than C++ because of the complication caused by the
 803 hierarchical overriding feature.

804 6 Related Work

805 We describe related work in four parts. We first discuss mainstream popular multiple
 806 inheritance models and then some specific models (e.g., C++ and C#) which are closest

to our work. Then we discuss related techniques used in **SELF**. Finally, we discuss the foundation and related work of our formalization.

6.1 Mainstream Multiple Inheritance Models

Multiple inheritance is a useful feature in object-oriented programming, although it is difficult to model and can cause various problems (e.g. the diamond problem [27, 29]). There are many existing languages/models that support multiple inheritance [10, 22, 5, 28, 17, 19, 20, 11, 2]. The mixin models [5, 11, 32, 2, 13] allow naming components that can be applied to various classes as reusable functionality units. However, the linearization (total ordering) of mixin inheritance cannot provide a satisfactory resolution in some cases and restricts the flexibility of mixin composition. Scala traits [22] are in fact linearized mixins and hence have the same problem as mixins.

Simplifying the mixins approach, traits [28, 9] draw a strong line between units of reuse and object factories. Traits act as units of reuse, containing functionality code. Classes, on the other hand, are assembled from traits and act as object factories. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the **default** keyword) inside interfaces, thus allowing for a restricted form of multiple inheritance. There are also proposals such as FeatherTrait Java [16] for extending Java with traits. Extensions [25, 26] to the original trait model exists with various advanced features, such as *renaming*. As discussed in Section 2, the renaming feature gives a workaround to the unintentional method conflicts problem. However, it breaks structural subtyping.

Malayeri and Aldrich proposed a model CZ [17] which aims to do multiple inheritance without the diamond problem. Inheritance is divided into two concepts: inheritance dependency and implementation inheritance. Using a combination of **requires** and **extends**, a program with diamond inheritance is transformed to one without diamonds. Moreover, fields and multiple inheritance can coexist. However untangling inheritance also untangles the class structure. In CZ, not only the number of classes but also the class hierarchy complexity increases.

The above-mentioned models/languages support multiple inheritance, focusing on diamond inheritance. They handle method conflicts in the same way, by simply disallowing two methods with the same signature from two different units to coexist. In contrast, our work provides mechanisms that allow methods with the same signatures, but different parents to coexist in a class. Disambiguation is possible in many cases by using both static and dynamic type information during method dispatching. In the cases where real ambiguity exists, **FHJ**'s type system can reject interface definitions and/or method calls statically.

6.2 Resolving Unintentional Method Conflicts

A few language implementations have realized the problem of unintentional conflicts and provide some support for it.

C++ model. C++ supports a very flexible inheritance model. C++ allows the existence of unintentional conflicts and users may specify a hierarchical path via casts for disambiguation, as discussed in Section 2. With virtual methods, dynamic dispatch is used and the method lookup algorithm will find the most specific method definition. A contribution of our work is to provide a minimal formal model of hierarchical dispatching, whereas C++ can be viewed as a real-world implementation. There are several formalizations [34, 24, 23] in the literature modeling various C++ features. However, as far as we know, there is no formal model that captures this aspect of the C++ method dispatching model. Apart from this, as discussed in


```

class A { public: virtual void m() {cout << "MA" << endl;}};
class B { public: virtual void m() {cout << "MB" << endl;}};
template<class C>
class MiddleMan : public C {
    void m() override final { m_impl(this); }
protected:
    virtual void m_impl(MiddleMan*) { return this->C::m(); }
};
class C : public MiddleMan<A>, public MiddleMan<B> {
private:
    void m_impl (MiddleMan<A>*) override {cout << "MA2" << endl;}
    void m_impl (MiddleMan<B>*) override {cout << "MB2" << endl;}
};
int main()
{
    C* c = new C();
    ((A*)c)->m();    //print "MA2"
    return 0;
}

```

■ **Figure 8** The *MiddleMan* approach.

852 Section 5.3, **FHJ** conservatively rejects some interface/class definitions that C++ accepts,
 853 and upcasts are never rejected since they are used for ambiguity resolution.

854 Although C++ supports hierarchical dispatching, it does not support hierarchical overriding.
 855 However, there are some possible workarounds that can mimick hierarchical overriding,
 856 including the *MiddleMan* approach³, the *interface classes* pattern as described in Section
 857 25.6 of [31], the *LotterySimulation* discussion in [30]. Since these workarounds share the same
 858 spirit, we will discuss in detail the *MiddleMan* approach, with the code shown in Figure 8.
 859 In this example, classes A and B are two classes that both define a method with the same
 860 name *m* unintentionally.

861 Class *MiddleMan*, as its name suggests, acts as a middle man between its class C and its
 862 parents A, B. *MiddleMan* defines a virtual method *m* that overrides a parent method *m* and
 863 delegates the implementation to another method *m_impl* that takes *this* as a parameter. C++
 864 supports method overloading, so that multiple *m_impl* methods with different parameter
 865 types can coexist. When defining class C, we specify the parents to be *MiddleMan<A>*,
 866 *MiddleMan* instead of A, B. In this way, programmers may define new versions of A.*m*
 867 and B.*m* in class C by providing the corresponding *m_impl* methods. Then in the client
 868 code, the method call ((A*)c)->m() will print out string "MA2", as expected. Although
 869 this workaround can help us defining partial method overrides to a certain extent, the
 870 drawbacks are obvious. Firstly, the approach is complex and requires the programmer to fully
 871 understand this approach. Moreover, the lack of direct syntax support makes *MiddleMan*
 872 code cumbersome to write. Finally, the approach is ad-hoc, meaning that the class *MiddleMan*
 873 shown in Figure 8 is not general enough to be used in other cases: more middle-men are
 874 needed if partial method overrides happen in other classes; and it is even worse when return
 875 types differ.

876 **C# explicit method implementations.** Explicit method implementations is a special
 877 feature supported by C#. As described in C# documentation [19], a class that implements an
 878 interface can explicitly implement a member of that interface. When a member is explicitly

³ <https://stackoverflow.com/questions/44632250/can-i-do-mimic-things-like-this-partial-override-in-c>

```

mixin HelloWorld of Object =
  new Object MainMatter()
  begin
    "Hello world".String.print();
  end;
end;
(new HelloWorld []).HelloWorld.MainMatter();

```

■ **Figure 9** Full-qualified name of method calls in Magda.

implemented, it can only be accessed through an instance of the interface. Explicit interface implementations allow an interface to inherit multiple interfaces that share the same member names and give each interface member a separate implementation.

Explicit interface member implementations have two advantages. Firstly, they allow interface implementations to be excluded from the public interface of a class. This is particularly useful when a class implements an internal interface that is of no interest to a consumer of that class or struct. Secondly, they allow disambiguation of interface members with the same signature. However, there are two critical differences to **FHJ**: (1) default method implementations are not allowed in C# interfaces; (2) there is only one level of conflicting method implementations at the class that implements the multiple parent interfaces. Further overriding of those methods is not possible in subclasses.

Languages using hygienicity. In NextGen/MixGen [1], HygJava [15] and Magda [3], *hygienicity* is proposed to deal with unintentional method conflicts. The idea is to give a method a unique identifier by prefixing the name with an unambiguous path. As shown in Figure 9, the prefix `HelloWorld` in method call `(new HelloWorld []).HelloWorld.MainMatter()` is mandatory. So writing programs in these languages is tedious if not supported by a specialized IDE, that aids filling prefix/method information. The advantage of this approach, compared to ours, is that it does not require any additional notion for method dispatching. Indeed the compilation strategy is simple, just by generating conventional code (say in Java or C++) with method names attached with prefixes. Unfortunately, the disadvantage is that some expressive power is lost. In particular *merging* methods arising from diamond inheritance is not possible because the methods have different prefixes. As shown in Figure 10, two methods `m` from different branches `A` and `B` cannot be overridden by the method `m` in `C` because they are regarded as unrelated methods, and `m` in `C` is just another new method that has nothing to do with `A.m` or `B.m`. The reason is that in these hygienic approaches, path names are used to distinguish different methods. In contrast, our model can deal with unintentional conflicts, as well as merged methods because our semantics is not simply based on prefixing. Instead, our model keeps the names of methods unchanged, and our direct operational semantics takes static and dynamic type information into account at runtime when doing method dispatching. Finally, the multiple inheritance model in Magda is based on Mixins, whereas **FHJ** is based on traits. Thus, Magda inherits all limitations of Mixins (such as the linearization problem, etc).

6.3 Hierarchical Dispatch in SELF

As we have discussed before, although the mix of static and dynamic dispatch is particularly useful under certain circumstances, it has received little research attention. In the prototype-based language **SELF** [6], inheritance is a basic feature. **SELF** does not include classes but instead allows individual objects to inherit from (or delegate to) other objects. Although it

```

mixin A of Object =
  new String m()
  begin
    return "A";
  end;
end;
mixin B of Object =
  new String m()
  begin
    return "B";
  end;
end;
mixin C of A, B =
  new String m()
  begin
    return "C";
  end;
end;

```

■ **Figure 10** Code in Magda.

is different from class-based languages, the multiple inheritance model is somewhat similar. The **SELF** language supports multiple (object) inheritance in a clever way. It not only develops the new inheritance relation with *prioritized parents* but also adopts *sender path tiebreaker rule* for method lookup. In **SELF** “if two slots with the same name are defined in equal-priority parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message, then that parent’s slot takes precedence over the other parent’s slot”. Similarly to our model, this sender path tiebreaker rule resolves ambiguities between unrelated slots. However, it is used in a prototype-based language setting and it does not support method hierarchical overriding as **FHJ** does.

6.4 Formalization Based on Featherweight Java

Featherweight Java (FJ) [14] is a minimal core calculus of the Java language, proposed by Igarashi et. al. There are many models built on Featherweight Java, including Feather-Trait [16], Featherweight defenders [12], Jx [21], Featherweight Scala [8], and so on. FJ provides the standard model of formalizing Java-like object-oriented languages and is easily extensible. In terms of formalization, the key novelty of our model is making use of various types (such as parameter types, method return types, etc) as upcasts along with various terms. As far as we know, this technique has not appeared in the literature before. This notion is of vital importance in our hierarchical dispatch algorithm, and it allows for a more precise subject-reduction theorem as discussed in Section 3.

7 Conclusion

This paper proposes **FHJ** as a formalized multiple inheritance model for unintentional method conflicts. Previous approaches either do not support unintentional method conflicts, thus have to compromise between code reuse and type safety, or do not fully support overriding in the presence of unintentional conflicts. To deal with unintentional method conflicts we introduce two key mechanisms: hierarchical dispatching and hierarchical overriding. Hierarchical dispatching is inspired by the mechanisms in C++. We provide a minimal formal model of hierarchical dispatching in **FHJ**. Such an algorithm makes use of both dynamic type

information and static information from either upcasts or parameters' information. It not only offers great code reuse like dynamic dispatch but also ensures unambiguity by our algorithm for method resolution. Additionally we introduce *hierarchical overriding* to allow conflicting methods in different branches to be individually overridden.

FHJ is formalized following the style of Featherweight Java and proved to be sound. A prototype interpreter is implemented in Scala. We believe that the formalization of hierarchical dispatching features is general and can be safely embedded in other OO models, so as to have support for the fork inheritance.

Our model can certainly be improved in some aspects. As discussed in Section 5, there are orthogonal and non-orthogonal features that can potentially be added to the design space. The future work relates to loosening the model without giving up its soundness, together with more exploration on supporting fields in the multiple inheritance setting.

References

- 1 Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 96–114, New York, NY, USA, 2003. ACM. URL: <http://doi.acm.org/10.1145/949305.949316>, doi: 10.1145/949305.949316.
- 2 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.
- 3 Viviana Bono, Jarek Kuśmierek, and Mauro Mulaturo. Magda: A new language for modularity. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 560–588, Berlin, Heidelberg, 2012. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_25, doi:10.1007/978-3-642-31057-7_25.
- 4 Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *PPPJ '14*, 2014.
- 5 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.
- 6 Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *Lisp Symb. Comput.*, 4(3), July 1991.
- 7 Steve Cook. Varieties of inheritance. In *OOPSLA '87 Panel P2*, 1987.
- 8 Vincent Cremet, François Garillot, Serguei Lenglet, and Martin Odersky. A core calculus for scala type checking. In *MFCS '06*, 2006.
- 9 Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
- 10 Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.
- 11 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98*, 1998.
- 12 Brian Goetz and Robert Field. Featherweight defenders: A formal model for virtual extension methods in java. <http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf>, 2012.
- 13 James Hendler. Enhancement for multiple-inheritance. In *OOPWORK '86*, 1986.
- 14 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):396–450, 2001.
- 15 Jarosław DM Kusmerek and Viviana Bono. Hygienic methods - introducing hyjava. *Journal of Object Technology*, 6(9):209–229, 2007.

- 990 16 Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java.
991 *ACM Trans. Program. Lang. Syst.*, 30(2):11, 2008.
- 992 17 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In
993 *OOPSLA '09*, 2009.
- 994 18 B Meyer. Eiffel: Programming for reusability and extendibility. *SIGPLAN Not.*, 22(2),
995 1987.
- 996 19 Microsoft. Csharp explicit interface member implementations document. [https://msdn.](https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx)
997 [microsoft.com/en-us/library/aa664591\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx), 2003.
- 998 20 David A. Moon. Object-oriented programming with flavors. In *OOPSLA '86*, 1986.
- 999 21 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via
1000 nested inheritance. In *OOPSLA '04*, 2004.
- 1001 22 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth,
1002 Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger.
1003 An overview of the scala programming language. Technical report, 2004.
- 1004 23 G. Ramalingam and Harini Srinivasan. A member lookup algorithm for c++. In *PLDI '97*,
1005 1997.
- 1006 24 Tahina Ramananandro. *Mechanized Formal Semantics and Verified Compilation for C++*
1007 *Objects*. PhD thesis, Université Paris-Diderot-Paris VII, 2012.
- 1008 25 John Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *FOOL/-*
1009 *WOOD '06*, 2006.
- 1010 26 John Reppy and Aaron Turon. Metaprogramming with traits. In *ECOOP '07*, 2007.
- 1011 27 Markku Sakkinen. Disciplined inheritance. In *ECOOP '89*, 1989.
- 1012 28 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits:
1013 Composable units of behaviour. In *ECOOP '03*, 2003.
- 1014 29 Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIG-*
1015 *PLAN OOPS Mess.*, 1995.
- 1016 30 Bjarne Stroustrup. *The design and evolution of C++*. Pearson Education India, 1994.
- 1017 31 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.
- 1018 32 Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal
1019 operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*,
1020 3(1):1–30, 1996.
- 1021 33 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless java.
1022 In *GPCE '16*, 2016.
- 1023 34 Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational seman-
1024 tics and type safety proof for multiple inheritance in c++. In *OOPSLA '06*, 2006.
- 1025 35 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*,
1026 115(1):38–94, 1994.
- 1027 36 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression
1028 problem. In *FOOL '05*, 2005.

A Appendix

A.1 Proofs

► **Lemma 5.** *If $\text{mbody}(\mathbf{m}, I_d, I_s) = (J, \overline{I_x} \ \overline{x}, I_e \ e_0)$, then $\overline{x} : \overline{I_x}, \text{this} : J \vdash e_0 : I_0$ for some $I_0 <: I_e$.*

Proof. By the definition of `mbody`, the target method \mathbf{m} is found in J . By the method typing rule (T-METHOD), there exists some $I_0 <: I_e$ such that $\overline{x} : \overline{I_x}, \text{this} : J \vdash e_0 : I_0$. ◀

► **Lemma 6** (Weakening). *If $\Gamma \vdash e : I$, then $\Gamma, x : J \vdash e : I$.*

Proof. Straightforward induction. ◀

► **Lemma 7** (Method Type Preservation). *If $\text{mbody}(\mathbf{m}, J, J) = (K, \overline{I_x} \ _, I_e \ _)$, then for any $I <: J$, $\text{mbody}(\mathbf{m}, I, J) = (K', \overline{I_x} \ _, I_e \ _)$.*

Proof. Since `mbody`(\mathbf{m}, J, J) is defined, by (T-INTF) we derive that `mbody`(\mathbf{m}, I, J) is also defined. Suppose that

$$\text{findOrigin}(\mathbf{m}, J, J) = \{I_0\}$$

$$\text{findOverride}(\mathbf{m}, J, I_0) = \{K\}$$

$$\text{findOrigin}(\mathbf{m}, I, J) = \{I'_0\}$$

$$\text{findOverride}(\mathbf{m}, I, I'_0) = \{K'\}$$

Below we use $I[\mathbf{m} \uparrow J]$ to denote the type of method \mathbf{m} defined in I that overrides J . We have to prove that $K'[\mathbf{m} \uparrow I'_0] = K[\mathbf{m} \uparrow I_0]$. Two facts:

- A. By (T-INTF), `canOverride` ensures that an override between any two original methods preserves the method type. Formally,

$$I_1 <: I_2 \Rightarrow I_1[\mathbf{m} \uparrow I_1] = I_2[\mathbf{m} \uparrow I_2]$$

- B. By (T-METHOD) and (T-ABSMETHOD), any partial override also preserves method type. Formally,

$$I_1 <: I_2 \Rightarrow I_1[\mathbf{m} \uparrow I_2] = I_2[\mathbf{m} \uparrow I_2]$$

By definition of `findOverride`, $K <: I_0, K' <: I'_0$. By Fact B,

$$K[\mathbf{m} \uparrow I_0] = I_0[\mathbf{m} \uparrow I_0] \quad K'[\mathbf{m} \uparrow I'_0] = I'_0[\mathbf{m} \uparrow I'_0]$$

Hence it suffices to prove that $I'_0[\mathbf{m} \uparrow I'_0] = I_0[\mathbf{m} \uparrow I_0]$. Actually when calculating `findOrigin`(\mathbf{m}, J, J), by the definition of `findOrigin` we know that $I_0 <: J$ and $I_0[\mathbf{m} \text{ override } I_0]$ is defined. So when calculating `findOrigin`(\mathbf{m}, I, J) with $I <: J$, I_0 should also appear in the set before pruned, since the conditions are again satisfied. But after pruning, only I'_0 is obtained, by definition of `prune` it implies $I'_0 <: I_0$. By Fact A, the proof is done. ◀

► **Lemma 8** (Term Substitution Preserves Typing). *If $\Gamma, \overline{x} : \overline{I_x} \vdash e : I$, and $\Gamma \vdash \overline{y} : \overline{I_x}$, then $\Gamma \vdash [\overline{y}/\overline{x}]e : I$.*

1050 **Proof.** We prove by induction. The expression e has the following cases:

1051 **Case Var.** Let $e = x$. If $x \notin \bar{x}$, then the substitution does not change anything.

1052 Otherwise, since \bar{y} have the same types as \bar{x} , it immediately finishes the case.

Case Invk. Let $e = e_0.m(\bar{e})$. By (T-INVK) we can suppose that

$$\Gamma, \bar{x} : \bar{I}_x \vdash e_0 : I_0 \quad \text{mbody}(m, I_0, I_0) = (_, \bar{J} _, I _)$$

$$\Gamma, \bar{x} : \bar{I}_x \vdash \bar{e} : \bar{I}_e \quad \bar{I}_e <: \bar{J} \quad \Gamma, \bar{x} : \bar{I}_x \vdash e : I$$

By induction hypothesis,

$$\Gamma \vdash [\bar{y}/\bar{x}]e_0 : I_0 \quad \Gamma \vdash [\bar{y}/\bar{x}]\bar{e} : \bar{I}_e$$

1053 Again by (T-INVK), $\Gamma \vdash [\bar{y}/\bar{x}]e : I$.

1054 **Case New.** Straightforward.

1055 **Case Anno.** Straightforward by induction hypothesis and (T-ANNO).
1056 ◀

1057 A.1.1 Proof for Theorem 1

Proof.

Case S-Invk. Let

$$e = ((J)\text{new } I()).m(\bar{v}) \quad \Gamma \vdash e : I_e$$

$$e' = (I_{e_0})[(\bar{I}_x)\bar{v}/\bar{x}, (I_0)\text{new } I()/\text{this}]e_0$$

$$\text{mbody}(m, I, J) = (I_0, \bar{I}_x \bar{x}, I_{e_0} e_0)$$

Since $\text{mbody}(m, I, J)$ is defined, the definition of mbody ensures that $I <: J$. And since e is well-typed, by (T-INVK),

$$\Gamma \vdash \bar{v} : \bar{I}_v \quad \bar{I}_v <: \bar{I}_x$$

By the rules (T-ANNO) and (T-NEW),

$$\Gamma \vdash (\bar{I}_x)\bar{v} : \bar{I}_x \quad \Gamma \vdash (I_0)\text{new } I() : I_0$$

On the other hand, by Lemma 5,

$$\bar{x} : \bar{I}_x, \text{this} : I_0 \vdash e_0 : I'_{e_0} \quad I'_{e_0} <: I_{e_0}$$

By Lemma 6,

$$\Gamma, \bar{x} : \bar{I}_x, \text{this} : I_0 \vdash e_0 : I'_{e_0}$$

Hence by Lemma 8, the substitution preserves typing, thus

$$\Gamma \vdash [(\bar{I}_x)\bar{v}/\bar{x}, (I_0)\text{new } I()/\text{this}]e_0 : I'_{e_0}$$

1058 Since $I'_{e_0} <: I_{e_0}$, the conditions of (T-ANNO) are satisfied, hence $\Gamma \vdash e' : I_{e_0}$. Now
1059 we only need to prove that $I_{e_0} = I_e$. Since I_{e_0} is from $\text{mbody}(m, I, J)$, whereas I_e is from
1060 $\text{mbody}(m, J, J)$, by the rule (T-INVK) on e . Since $I <: J$, by Lemma 7, $I_{e_0} = I_e$.

1061 **Case C-Receiver.** Straightforward induction.

1062 **Case C-Args.** Straightforward induction.

1063 **Case C-StaticType.** Immediate by (T-ANNO).

1064 **Case C-FReduce.** Immediate by (T-ANNO) and induction.

1065 **Case C-AnnoReduce.** Immediate by (T-ANNO) and transitivity of $<:$.
1066 ◀

1067 **A.1.2 Proof for Theorem 2**

Proof. Since e is well-typed, by (T-INVK) and (T-ANNO) we know that

$$I <: J, \text{ and } \text{mbody}(m, J, J) \text{ is defined}$$

1068 By (T-INTF), $\text{mbody}(m, I, J)$ is also defined, and the type checker ensures the expected
1069 number of arguments.

On the other hand, since $I <: J$, by the definition of findOrigin ,

$$\text{findOrigin}(m, I, J) \subseteq \text{findOrigin}(m, I, I)$$

1070 By (T-NEW), $\text{canInstantiate}(I) = \text{True}$. By the definition of canInstantiate , any
1071 $J_0 \in \text{findOrigin}(m, I, I)$ satisfies that $\text{findOverride}(m, I, J_0)$ contains only one interface, in
1072 which the m that overrides J_0 is a concrete method. Therefore $\text{mbody}(m, I, J)$ also provides a
1073 concrete method, which finishes the proof. ◀

1074 **A.1.3 Proof for Theorem 4**

1075 **Proof.** The Proof is done by induction on a derivation of $t \rightarrow t'$, following the book *TAPL*.

- 1076 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (S-INVK), then we know that t has the
1077 form $((J)\text{new } I()) . m(\bar{v})$ with I, J, m determined. Now it is obvious that the last rule in the
1078 derivation of $t \rightarrow t''$ should also be (S-INVK) with the same I, J, m . Since $\text{mbody}(m, I, J)$
1079 is a *function* that given the same input will calculate the same result, we know the two
1080 induction results are the same, thus $t' = t''$ is immediately proved.
- 1081 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (C-RECEIVER), then t has the form
1082 $e_0 . m(\bar{e})$ and $e_0 \rightarrow e'_0$. Since e_0 is not a value, the last rule used in $t \rightarrow t''$ has to be
1083 (C-RECEIVER) (other rules do not match) too. Assume in the reduction $t \rightarrow t''$, $e_0 \rightarrow e''_0$,
1084 thus $e'_0 . m(\bar{e}) = e''_0 . m(\bar{e})$. Thus, $t' = t''$ proved.
- 1085 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (C-STATIC TYPE), then t is fixed to
1086 be $\text{new } I()$. The last rule used in $t \rightarrow t''$ has to be (C-STATIC TYPE), and obviously,
1087 $t' = t'' = (I)\text{new } I()$.
- 1088 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (C-FREDUCE), then t has the form
1089 $(I)e$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (C-STATIC TYPE) because it
1090 requires t to be $\text{new } I()$; it can neither be (C-ANNOREDUCE) because it requires t to
1091 be $(I)((J)\text{new } K())$ where $(J)\text{new } K()$ is already a value. So the last rule used in $t \rightarrow t''$
1092 can only be (C-FREDUCE) (other rules do not match). Assume in the reduction $t \rightarrow t''$,
1093 $e \rightarrow e''$, and $(I)e \rightarrow (I)e''$. By induction hypothesis, $e' = e''$, thus $t' = t''$ proved.
- 1094 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (C-ANNOREDUCE), then the form
1095 of t is fixed to be $(I)((J)\text{new } K())$. Since $(I)((J)\text{new } K())$ is not reducible, the rule (C-
1096 FREDUCE) does not apply. The only rule applies in $t \rightarrow t''$ is (C-ANNOREDUCE). Thus
1097 $t' = t'' = (I)\text{new } K()$ proved.
- 1098 ■ If the last rule used in the derivation of $t \rightarrow t'$ is (C-ARGS), then t has the form
1099 $v.m(\dots, e, \dots)$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (S-INVK) because it
1100 requires all arguments to be values. Thus only (C-ARGS) applies to $t \rightarrow t''$. Assume in
1101 the reduction $t \rightarrow t''$, $e \rightarrow e''$. By induction hypothesis, $e' = e''$, thus $v.m(\dots, e', \dots) =$
1102 $v.m(\dots, e'', \dots)$, thus $t' = t''$ proved.

1103 ◀