FHJ: A Formal Model for Hierarchical Dispatching and Overriding

John Q. Open[1] and Joan R. Access[2]

1 **Dummy University Computing Laboratory, Address/City, Country**
open@dummyuniversity.org
2 **Department of Informatics, Dummy College, Address/City, Country**
access@dummycollege.org

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 ***F**eatherweight **H**ierarchical **J**ava* (**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.

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Dummy keyword – please provide 1–5 keywords

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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 the start. Scala adapts the ideas from traits [24, 7, 13] and mixins [3, 9, 27, 1, 11] to offer a disciplined form of multiple inheritance. Java 8 offers a simple variant of traits, disguised of interfaces with default methods [10].

A reason why programming languages have resisted to multiple inheritance in the past is that, as Cook [5] 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* [23, 25] (also known as "fork-join

inheritance" [23]). 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 [14], 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 [28, 14, 26].

In contrast to diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [24]. 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. 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 methods, 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 *triangle 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* [31].

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 triangle inheritance and resolves the ambiguity by specifying the expected path by *upcasts*. However, neither the C# or 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 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

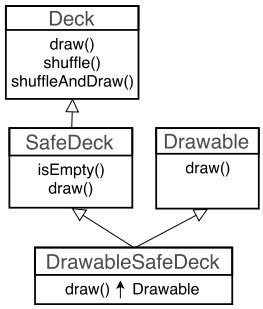


Figure 1 `DrawableSafeDeck`: an illustration of hierarchical overriding.

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 and 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 overriding *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 [12], together with theorems and proofs for type soundness. 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.
- **Prototype implementation**[1]**:** a simple implementation of **FHJ** interpreter in Scala. The implementation can type-check and run variants of all the examples shown in this paper.

2 A Running Example: Drawable Deck

This section illustrates the problem of unintentional method conflicts, together with the features of our model for addressing this issue, by a simple running example. In the following text we will introduce three problems one by one and have a discussion on possible

[1] https://github.com/YanlinWang/MIM/tree/master/Calculus

workarounds and our solutions. Problems 1 and 2 are related to hierarchical dispatching, and in C++ it is possible to have similar solutions to both problems. Hence it is important to emphasize that, with respect to hierarchical dispatching, our model is not a novel mechanism. Instead, inspired by the C++ solutions, our contribution is formalizing a minimal calculus of this feature together with a proof of type soundness. However, for the final problem, there is no satisfactory approach in existing languages, thus what we propose is a novel feature (hierarchical overriding) with the corresponding formalization of that feature.

In the rest of the paper, we use a Java-like syntax for programs. All types are defined with the keyword "**interface**"; the concept is closely related to Java 8 interfaces with default methods [2] and traits. In short, an interface in our model has the following characteristics:

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

2.1 Problem 1: Basic Unintentional Method Conflicts

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

```
interface Deck {
  void draw() { // draws a card from the Deck
    println("Draw a card.");
  }
}
interface Drawable {
  void draw() { // create a blank canvas
    JFrame frame = new JFrame();
    frame.setVisible(true);
  }
}
```

In `Deck`, `draw` uses `println`, which is a library function. The two `draw` methods can have different return types, but for simplicity, the return types are both **void** here. Note that, similarly to Featherweight Java [12], **void** is unsupported in our formalization. We could have also defined an interface called `Void` and return an object of that type instead. To be concise, 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, and hence the compiler throws an error on `DrawableDeck` about the conflict.

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.

2.1.0.1 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 in traits.* 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.

2.1.0.2 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() {...} }
interface DrawableDeck extends Drawable, Deck {}
// main program
((Deck) new DrawableDeck()).draw() // calls Deck.draw
// (new DrawableDeck()).draw() // this call is ambiguous and rejected
```

In our language, the main program is merely an expression. The above cast indicates that we expect to invoke the `draw` method from the branch `Deck`. Similarly, we could have used an upcast to `Drawable` to call the `draw` method from `Drawable`. Without the cast, the call would be ambiguous and **FHJ**'s type system would reject it.

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

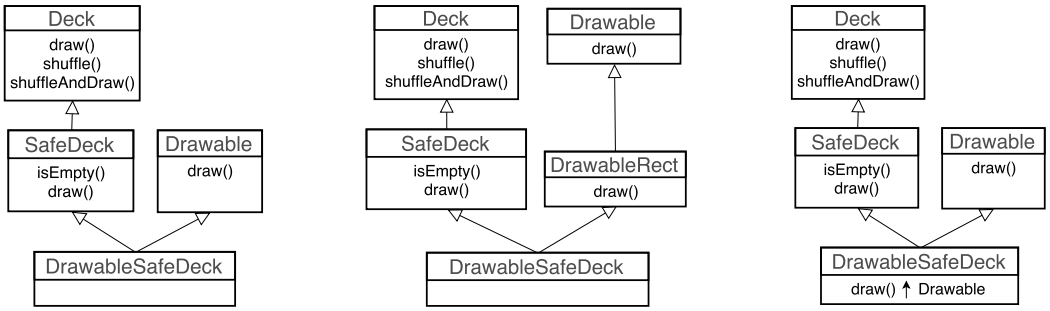


Figure 2 UML diagrams for 3 variants of `DrawableSafeDeck`.

2.2 Problem 2: Dynamic Dispatching

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

```
interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}
```

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

```
interface SafeDeck extends Deck {
  boolean isEmpty() {...}
  void draw() { // overriding
    if (isEmpty()) println("The deck is empty.");
    else println("Draw a card");
  }
}
```

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

```
interface DrawableSafeDeck extends Drawable, SafeDeck {}
// main program
new DrawableSafeDeck().shuffleAndDraw()
```

Indeed, using a reduction model similar to FJ [12] (without static types tracked), the reduction steps would roughly be:

```
    new DrawableSafeDeck().shuffleAndDraw()
-> new DrawableSafeDeck().shuffle(); new DrawableSafeDeck().draw()
-> ...
-> new DrawableSafeDeck().draw()
-> <<error: ambiguous call!!!>>
```

When the `DrawableSafeDeck` object calls `shuffleAndDraw`, the implementation in `Deck` is dispatched. But then `shuffleAndDraw` invokes "**this**`.draw()`", and at this point, the receiver is replaced by the object **new** `DrawableSafeDeck()`. From the perspective of `DrawableSafeDeck`, the `draw` method seems to be ambiguous since `DrawableSafeDeck` inherits two `draw` methods from both `SafeDeck` and `Drawable`. But ideally we would like `shuffleAndDraw` to invoke `SafeDeck.draw` because they belong to the same class hierarchy branch.

2.2.0.1 FHJ's solution:

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

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

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

```
    new DrawableSafeDeck().shuffleAndDraw()
 -> ((DrawableSafeDeck) new DrawableSafeDeck()).shuffleAndDraw()
 -> ((Deck) new DrawableSafeDeck()).shuffle(); ((Deck) new DrawableSafeDeck()).draw()
 -> ...
 -> ((Deck) new DrawableSafeDeck()).draw()
 -> ... // SafeDeck.draw
```

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

2.3.0.1 Potential solutions/workarounds in existing languages:

Unfortunately in all languages we have known (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.
- Separate overriding is required to come after the triangle inheritance, especially when the implementation needs functionality from both parents. In the above code, we have assumed that the overriding is unrelated to `Deck`. But when the drawing relies on some information of the `Deck` object, we have to either introduce field methods for delegation or change the signature of `draw` to take a parameter. Either way introduces unnecessary complexity and affects extensibility.

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

2.3.0.2 FHJ's solution:

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

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

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

2.3.0.3 Terminology:

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

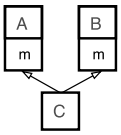
A special rule for hierarchical overriding is: it can only refine *original* methods, and cannot jump over original methods with the same signature. For instance, writing `"void draw() override Deck {...}"` is disallowed in `DrawableSafeDeck`, because existing two branches are `Drawable.draw` and `SafeDeck.draw`,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

2.3.0.4 A peek at the hierarchical dispatching algorithm:

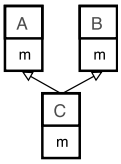
In **FHJ**, triangle inheritance allows several original methods (branches) to coexist, and hierarchical dispatch first finds the most specific original method (branch), then it finds the most specific hierarchical overriding on that branch.

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, $\texttt{mbody}(\mathsf{m}, \mathsf{X}, \mathsf{Y}) = (\mathsf{Z}, ...)$ reflects that the source code `((Y)new X()).m()` calls `Z.m` in the runtime. It is undefined when method dispatch is ambiguous.

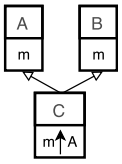
In Figure 3, (a) is the base case for unintentional conflicts, namely the triangle inheritance. (b) uses overriding to explicitly merge the conflicting methods. (c) represents hierarchical overriding. (d) and (e) are two base cases of diamond inheritance in **FHJ** and the definition of each C is rejected because T is an ambiguous parent to C. (f) gives a common solution



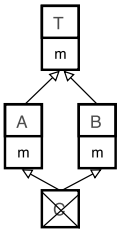
(a) $\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{A}) = (\mathtt{A}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{B}) = (\mathtt{B}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{C}) = \texttt{undefined}$



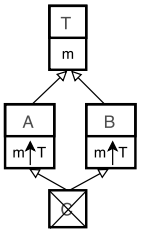
(b) $\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{A}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{B}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{C}) = (\mathtt{C}, ...)$



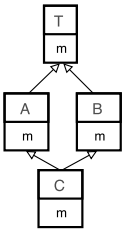
(c) $\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{A}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{B}) = (\mathtt{B}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{C}) = \texttt{undefined}$



(d) $\texttt{mbody}(\mathtt{m}, \mathtt{T}, \mathtt{T}) = (\mathtt{T}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{T}) = \texttt{undefined}$
C rejected by type-check



(e) $\texttt{mbody}(\mathtt{m}, \mathtt{T}, \mathtt{T}) = (\mathtt{T}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{T}) = \texttt{undefined}$
C rejected by type-check



(f) $\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{T}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{A}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{B}) = (\mathtt{C}, ...)$
$\texttt{mbody}(\mathtt{m}, \mathtt{C}, \mathtt{C}) = (\mathtt{C}, ...)$

Figure 3 Examples in **FHJ**. "$\mathtt{m} \uparrow \mathtt{A}$" stands for hierarchical overriding "$\mathtt{m}$ **override** $\mathtt{A}$".

to the diamond as in Java or traits, which is to explicitly override $\mathtt{A.m}$ and $\mathtt{B.m}$ in $\mathtt{C}$. In the last three examples, conflicting methods $\mathtt{A.m}$ and $\mathtt{B.m}$ should be viewed as intentional conflicts, as they come from the same source $\mathtt{T}$.

3 Formalization

In this section, we present a formal model called **FHJ** (***F**eatherweight **H**ierarchical **J**ava*), following the similar style of Featherweight Java [12]. **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.

3.1.0.1 Notations

The metavariables $\mathtt{I}, \mathtt{J}$ range over interface names; $\mathtt{x}$ ranges over variables; $\mathtt{m}$ ranges over method names; $\mathtt{e}$ ranges over expressions; and $\mathtt{M}$ ranges over method declarations. Following Featherweight Java, we assume that the set of variables includes the special variable `this`,

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

Figure 4 Syntax of **FHJ**.

which cannot be used as the name of an argument to a method. We use the same conventions as FJ; we write $\overline{\text{I}}$ as shorthand for a possibly empty sequence $\text{I}_1, ..., \text{I}_n$, which may be indexed by I_i; and write $\overline{\text{M}}$ as shorthand for $\text{M}_1..\text{M}_n$ (with no commas). We also abbreviate operations on pairs of sequences in an obvious way, writing $\overline{\text{I}}\ \overline{x}$ for $\text{I}_1\ x_1, ..., \text{I}_n\ x_n$, where n is the length of $\overline{\text{I}}$ and $\overline{x}$.

3.1.0.2 Interfaces

In order to achieve multiple inheritance, an interface can have a set of parent interfaces, where such a set can be empty. The interface declaration `interface` I `extends` $\overline{\text{I}}$ $\{\overline{\text{M}}\}$ introduces an interface named I with parent interfaces $\overline{\text{I}}$ and a suite of methods $\overline{\text{M}}$. The methods of I may either override methods that are already defined in $\overline{\text{I}}$ or add new functionality special to I, we will illustrate this in more detail later.

3.1.0.3 Methods

Original methods and hierarchically overriding methods share the same syntax in our model for simplicity. The concrete method declaration I $m(\overline{I_x}\ \overline{x})$ `override` J $\{$`return` $e;\}$ introduces a method named m with result type I, parameters $\overline{x}$ of type $\overline{I_x}$ 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; otherwise it is a hierarchical overriding method. The definition of abstract methods is written as I $m(\overline{I_x}\ \overline{x})$ `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.

3.1.0.4 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.

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

$$\boxed{\mathrm{I} <: \mathrm{J}} \qquad \mathrm{I} <: \mathrm{I}$$

$$\frac{\mathrm{I} <: \mathrm{J} \qquad \mathrm{J} <: \mathrm{K}}{\mathrm{I} <: \mathrm{K}} \qquad \frac{\texttt{interface I extends } \overline{\mathrm{I}}\ \{\overline{\mathrm{M}}\}}{\forall \mathrm{I}_i \in \overline{\mathrm{I}}, \mathrm{I} <: \mathrm{I}_i}$$

$$\boxed{\Gamma \vdash e : \mathrm{I}} \qquad \text{(T-Var) } \Gamma \vdash x : \Gamma(x)$$

$$\text{(T-Invk) } \frac{\Gamma \vdash e_0 : \mathrm{I}_0 \qquad \texttt{mbody}(m, \mathrm{I}_0, \mathrm{I}_0) = (\mathrm{K}, \overline{\mathrm{J}}\ \overline{x}, \mathrm{I}\ \_) \qquad \Gamma \vdash \overline{e} : \overline{\mathrm{I}} \qquad \overline{\mathrm{I}} <: \overline{\mathrm{J}}}{\Gamma \vdash e_0.m(\overline{e}) : \mathrm{I}}$$

$$\text{(T-New) } \frac{\texttt{interface I extends } \overline{\mathrm{I}}\ \{\overline{\mathrm{M}}\} \qquad \texttt{canInstantiate}(\mathrm{I})}{\Gamma \vdash \texttt{new I}() : \mathrm{I}}$$

$$\text{(T-Anno) } \frac{\Gamma \vdash e : \mathrm{I} \qquad \mathrm{I} <: \mathrm{J}}{\Gamma \vdash (\mathrm{J})e : \mathrm{J}}$$

$$\text{(T-Method) } \frac{\begin{array}{c}\mathrm{I} <: \mathrm{J} \qquad \texttt{findOrigin}(m, \mathrm{I}, \mathrm{J}) = \{\mathrm{J}\} \\ \texttt{mbody}(m, \mathrm{J}, \mathrm{J}) = (\mathrm{K}, \overline{\mathrm{I}_x}\ \overline{x}, \mathrm{I}_e\ \_) \qquad \overline{x} : \overline{\mathrm{I}_x}, \texttt{this} : \mathrm{I} \vdash e_0 : \mathrm{I}_0 \qquad \mathrm{I}_0 <: \mathrm{I}_e\end{array}}{\mathrm{I}_e\ m(\overline{\mathrm{I}_x}\ \overline{x})\ \texttt{override J}\ \{\texttt{return } e_0;\}\ \text{OK IN I}}$$

$$\text{(T-AbsMethod) } \frac{\mathrm{I} <: \mathrm{J} \quad \texttt{findOrigin}(m, \mathrm{I}, \mathrm{J}) = \{\mathrm{J}\} \qquad \texttt{mbody}(m, \mathrm{J}, \mathrm{J}) = (\mathrm{K}, \overline{\mathrm{I}_x}\ \overline{x}, \mathrm{I}_e\ \_)}{\mathrm{I}_e\ m(\overline{\mathrm{I}_x}\ \overline{x})\ \texttt{override J}\ ;\ \text{OK IN I}}$$

$$\text{(T-Intf) } \frac{\begin{array}{c}\overline{\mathrm{M}}\ \text{OK IN I} \\ \forall \mathrm{J} >: \mathrm{I} \text{ and } m, \texttt{mbody}(m, \mathrm{J}, \mathrm{J}) \text{ is defined} \Rightarrow \texttt{mbody}(m, \mathrm{I}, \mathrm{J}) \text{ is defined} \\ \forall \mathrm{J} >: \mathrm{I} \text{ and } m, \mathrm{I}[m\ \texttt{override I}] \text{ and } \mathrm{J}[m\ \texttt{override J}] \text{ defined} \Rightarrow \texttt{canOverride}(m, \mathrm{I}, \mathrm{J})\end{array}}{\texttt{interface I extends } \overline{\mathrm{I}}\ \{\overline{\mathrm{M}}\}\ \text{OK}}$$

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

3.2 Subtyping and Typing Rules

3.2.0.1 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 symmetric, reflexive and transitive.

3.2.0.2 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 $\texttt{mbody}(m, \mathrm{I}_0, \mathrm{I}_0)$ 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.

In (T-Invk), as the compilation should not be aware of the dynamic type, it only requires that invoking m is valid for the static type of the receiver. The result of `mbody` contains the interface that provides the most specific implementation, the parameters and the return type. We use underscore for the return expression, implying that an empty return expression from an abstract method is acceptable.

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

(T-Method) is more interesting since a method can either be an original method or a hierarchical overriding, though they share the same syntax and method typing rule. `findOrigin`(m, I, J) is a fundamental function, used to find "the most specific interfaces that are above I and along path J, and originally defines m" (see Section 4 for full definition). By "most specific interfaces", it implies that the inherited supertypes are excluded. Thus the condition `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 `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 `override` I] is defined if I originally defines m, and `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: `mbody`(m, J, J) is defined but `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 forms 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:

```
interface T           { T m() override T { return new T(); } }
interface A extends T { T m() override T { return new A(); } }
interface B extends T { T m() override T { return new B(); } }
interface C extends A, B {}
((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, `mbody`(m, T, T) will refer to T.m, but `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 `((A)new D()).m()` is one example of triggering ambiguity, but **FHJ** simply rejects the definition of C. To resolve the issue, the programmer

need 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 Congruence

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

3.3.0.1 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)`new` I() indicates the dynamic type I together with the static type J. Therefore `mbody`$(\mathtt{m}, \mathtt{I}, \mathtt{J})$ 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 by 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.

3.3.0.2 Congruence Rules

(C-Receiver), (C-Args) and (C-FReduce) are natural congruence rules on receivers, arguments, and cast-expressions, respectively. (C-StaticType) 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 Method Lookup Algorithm in `mbody`

`mbody`$(\mathtt{m}, \mathtt{I_d}, \mathtt{I_s})$ denotes the method body lookup function. We use $\mathtt{I_d}, \mathtt{I_s}$, since `mbody` is usually invoked by a receiver of a method m, with its dynamic type $\mathtt{I_d}$ and static type $\mathtt{I_s}$. 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.

$$\text{(S-Invk)}\ \frac{\texttt{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)}{((J)\texttt{new}\ I())\texttt{.}m(\overline{v}) \to (I_e)[\overline{(I_x)v}/\overline{x}, (I_0)\texttt{new}\ I()/\texttt{this}]e_0}$$

$$\text{(C-Receiver)}\ \frac{e_0 \to e_0'}{e_0\texttt{.}m(\overline{e}) \to e_0'\texttt{.}m(\overline{e})} \qquad \text{(C-Args)}\ \frac{e \to e'}{e_0\texttt{.}m(\ldots, e, \ldots) \to e_0\texttt{.}m(\ldots, e', \ldots)}$$

$$\text{(C-StaticType)}\ \texttt{new}\ I() \to\ (I)\texttt{new}\ I()$$

$$\text{(C-FReduce)}\ \frac{e \to e' \qquad e \neq \texttt{new}\ J()}{(I)e \to\ (I)e'}$$

$$\text{(C-AnnoReduce)}\ (I)((J)\texttt{new}\ K()) \to (I)\texttt{new}\ K()$$

Figure 6 Small-step semantics.

▷ *Definition of* $\texttt{mbody}(m, I_d, I_s)$:

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

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

4.2 Finding the Most Specific Origin: `findOrigin`

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

▷ *Definition of* $\texttt{findOrigin}(m, I, J)$:

- $\texttt{findOrigin}(m, I, J) = \texttt{prune}(\texttt{origins})$
  with: $\texttt{origins} = \{K \mid I <: K,\ \text{and}\ K <: J\ \vee\ J <: K,$
  $\text{and}\ K[m\ \texttt{override}\ K]\ \text{is defined}\}$

By the definition, an interface belongs to $\texttt{findOrigin}(m, I, J)$ if and only if:

- It originally defines $\mathtt{m}$;
- It is a supertype of I (including I);
- It is either a supertype or a subtype of J (including J);
- Any subtype of it does not belong to the same result set because of `prune`.

4.3 Finding the Most Specific Overriding: `findOverride`

The `findOrigin` function only focuses on original method implementations, all the hierarchical overriding methods are omitted during that step. On the other hand, $\texttt{findOverride}(\mathtt{m}, \mathsf{I}, \mathsf{J})$ has the assumption that J defines an original $\mathtt{m}$, and this function tries to find the interfaces with the most specific implementations that hierarchically overrides such an $\mathtt{m}$. Formally,

▷ *Definition of* $\texttt{findOverride}(\mathtt{m}, \mathsf{I}, \mathsf{J})$:

- $\texttt{findOverride}(\mathtt{m}, \mathsf{I}, \mathsf{J}) = \texttt{prune}(\mathit{overrides})$
  with: $\mathit{overrides} = \{\mathsf{K} \mid \mathsf{I} <: \mathsf{K},\ \mathsf{K} <: \mathsf{J} \text{ and } \mathsf{K}[\mathtt{m}\ \texttt{override}\ \mathsf{J}] \text{ is defined}$

By the definition, an interface belongs to $\texttt{findOverride}(\mathtt{m}, \mathsf{I}, \mathsf{J})$ if and only if:

- it is between I and J (including I, J);
- it hierarchically overrides $\mathsf{J}.\mathtt{m}$;
- any subtype of it does not belong to the same set.

4.4 Other Auxiliaries

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

▷ *Definition of* $\mathsf{I}[\mathtt{m}\ \texttt{override}\ \mathsf{J}]$:

- $\mathsf{I}[\mathtt{m}\ \texttt{override}\ \mathsf{J}] = \mathsf{I}_e\ \mathtt{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ \{\texttt{return}\ e_0;\}$
  with: $\texttt{interface}\ \mathsf{I}\ \texttt{extends}\ \overline{\mathsf{I}}\ \{\mathsf{I}_e\ \mathtt{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ \{\texttt{return}\ e_0;\}\ldots\}$
- $\mathsf{I}[\mathtt{m}\ \texttt{override}\ \mathsf{J}] = \mathsf{I}_e\ \mathtt{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ ;$
  with: $\texttt{interface}\ \mathsf{I}\ \texttt{extends}\ \overline{\mathsf{I}}\ \{\mathsf{I}_e\ \mathtt{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ ;\ldots\}$

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

▷ *Definition of* $\texttt{prune}(\mathit{set})$:

- $\texttt{prune}(\mathit{set}) = \{\mathsf{I} \in \mathit{set} \mid \nexists \mathsf{J} \in \mathit{set} \setminus \mathsf{I}, \mathsf{J} <: \mathsf{I}\}$

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

▷ *Definition of* `canOverride`(m, I, J) :

• `canOverride`$(m, I, J) =$ `True`

with: $I[m$ `override` $I] = I_e\ m(\overline{I_x}\ \overline{x})$ `override` $I \ldots$

$J[m$ `override` $J] = I_e\ m(\overline{I_x}\ \overline{y})$ `override` $J \ldots$

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

▷ *Definition of* `canInstantiate`(I) :

• `canInstantiate`$(I) =$ `True`

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

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

`canInstantiate`(I) checks whether interface I can be instantiated by the keyword `new`. `findOrigin`(m, I, I) represents the set of branches I inherits on method m. I can be instantiated if and only if for every branch, the most specific implementation is unambiguous and non-abstract.

4.5 Properties

We present the type soundness of the model by a few theorems below, following the standard technique of subject reduction and progress proposed by Wright and Felleisen [30]. The proof, together with some lemmas, is presented in Appendix. Type soundness states that if an expression is well-typed, then after many reduction steps it must reduce to a value, and its annotation is the same as the static type of the original expression.

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

Proof. See Appendix **??**. ◀

▶ **Theorem 2** (Progress). *Suppose* e *is a well-typed expression, if* e *includes* $((J)$`new` $I())\,.m(\overline{v})$ *as a sub-expression, then* `mbody`$(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)$ *and* $\#(\overline{x}) = \#(\overline{v})$ *for some* I_0, $\overline{I_x}$, $\overline{x}$, I_e *and* e_0.

Proof. See Appendix **??**. ◀

▶ **Theorem 3** (Type Soundness). *If* $\emptyset \vdash e : I$ *and* $e \to^* e'$ *with* e' *a normal form, then* e' *is a value* v *with* $\emptyset \vdash v : I$.

Proof. Immediate from Theorem 1 and Theorem 2. ◀

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

Our theorems are stricter than those of Featherweight Java [12]. In FJ, the subject reduction theorem states that after a step of reduction, the type of an expression may change to a subtype due to subtyping. However, in **FHJ**, the type remains unchanged because we keep track of the static types and use them for casting during reduction.

5 Discussion

In this section, we will discuss the design space and reflect about some of the design decisions of our work. We relate our language to traits, Java interfaces as well as other languages. Furthermore, we discuss ways to improve our work.

Figure 7 Triangle 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 [12] does not include abstract methods because of the orthogonality to the core calculus. In traits, the similar idea is to use keywords like "**require**" for abstract method declarations [24]. 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 abstract 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 $\mathtt{m}$ into a single one. **FHJ** behaves differently from Java in both cases. In the triangle inheritance case (left), C will have two distinct abstract methods corresponding to $\mathsf{A.m}$ and $\mathsf{B.m}$. 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, that 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:

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid e.J_0@J_1 :: m(\overline{e})$$

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

$$\text{(S-StaticInvk)} \ \frac{J_0[m \ \texttt{override} \ J_1] = I_e \ m(\overline{I_x} \ \overline{x}) \ \texttt{override} \ J_1 \ \{\texttt{return} \ e_0;\}}{((J)\texttt{new} \ I()).J_0@J_1 :: m(\overline{v}) \to (I_e)[\overline{(I_x)v}/\overline{x}, (J_0)\texttt{new} \ I()/\texttt{this}]e_0}$$

A static invocation $e.J_0@J_1 :: m(\overline{e})$ aims at finding the method m in J_0 that hierarchically overrides J_1, thus $J_0[m \ \texttt{override} \ J_1]$ 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.

One typical non-orthogonal extension to **FHJ** could be to have fields. The design of **FHJ** can be viewed as a variant of Java 8 with default methods which allows for unintentional method conflicts. Like Java interfaces and traits, state is forbidden in **FHJ**. There are some inheritance models that also account for fields, such as C++ that uses virtual inheritance [8]. In our model, however, we can perhaps borrow the idea of *interface-based programming* [28], which models state with abstract state operations. This can be realized by extending our current model with static methods and anonymous classes from Java. However such an extension requires more thought, so we leave it to future work.

5.3 Loosening the Model: Reject Early or Reject Later?

FHJ rejects the following case of diamond inheritance:

```
interface A { void m() {...} }
interface B extends A { void m() {...} }
interface C extends A { void m() {...} }
interface D extends B, C {}
```

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

Nonetheless, C++ accepts the definition of D, but forbids later upcasts from D to A because of ambiguity. Our language is more conservative on definitions of interfaces compared to C++, but on the upside, upcasts are not rejected. We could also loosen the model to accept definitions such as `D`, and perform ambiguity check on upcasts and other expressions. However, we would need to handle more cases than C++. For instance, in the above program assume that B.m and C.m are hierarchical overrides on A. Then such a case is still a diamond.

6 Related Work

We describe related work in four parts. We first discuss mainstream popular multiple inheritance models and the specific models (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 [23, 25]). There are many existing languages/models that support multiple inheritance [8, 18, 3, 24, 14, 15, 16, 9, 1]. The mixin models [3, 9, 27, 1, 11] 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 [18] are in fact linearized mixins and hence have the same problem as mixins.

Simplifying the mixins approach, traits [24, 7] 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 [13] for extending Java with traits. Extensions [21, 22] 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 [14] 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 method lookup algorithm will find the most specific method definition. A contribution of our work is

```
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.

to provide a minimal formal model of hierarchical dispatching, whereas C++ can be viewed as a real-world implementation. There are several formalizations [29, 20, 19] 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 Section 5.3, **FHJ** conservatively rejects some interface/class definitions that C++ accepts, and upcasts are never rejected since they are used for ambiguity resolution.

Although C++ supports hierarchical dispatching, it does not support hierarchical overriding. However, there is a possible workaround[3] that mimicks hierarchical overriding. We call it the *MiddleMan* approach, as shown in Figure 8. In this example, classes `A` and `B` are two classes that both define a method with the same name `m` unintentionally.

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

C# Explicit method implementations. Explicit method implementations is a special feature supported by C#. As described in C# documentation [15], a class that implements an 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-likes-this-partial-override-in-c

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.

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** [4], 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 is different from class-based languages, the multiple inheritance model is somehow 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 over 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) [12] 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 [13], Featherweight defenders [10], Jx [17], Featherweight Scala [6], 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 triangle 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 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.

2 Viviana Bono, Enrico Mensa, and Marco Naddeo. Trait-oriented programming in java 8. In *PPPJ '14*, 2014.

3 Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90*, 1990.

4 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.

5 S. COOK. Oopsla '87 panel p2: Varieties of inheritance. In *OOPSLA '87 Addendum To The Proceedings*, 1987.

6 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for scala type checking. In *MFCS '06*, 2006.

7 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.

8 Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, 1990.

9 Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL '98*, 1998.

10 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.

11 James Hendler. Enhancement for multiple-inheritance. In *OOPWORK '86*, 1986.

12 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.

13 Luigi Liquori and Arnaud Spiwack. Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2):11, 2008.

14 Donna Malayeri and Jonathan Aldrich. Cz: Multiple inheritance without diamonds. In *OOPSLA '09*, 2009.

15 Microsoft. Csharp explicit interface member implementations document. `https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx`, 2003.

16 David A. Moon. Object-oriented programming with flavors. In *OOPSLA '86*, 1986.

17 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04*, 2004.

18 Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

19 G. Ramalingam and Harini Srinivasan. A member lookup algorithm for c++. In *PLDI '97*, 1997.

20 Tahina Ramananandro. *Mechanized Formal Semantics and Verified Compilation for C++ Objects*. PhD thesis, Université Paris-Diderot-Paris VII, 2012.

21 John Reppy and Aaron Turon. A foundation for trait-based metaprogramming. In *FOOL/-WOOD '06*, 2006.

22 John Reppy and Aaron Turon. Metaprogramming with traits. In *ECOOP '07*, 2007.

23 Markku Sakkinen. Disciplined inheritance. In *ECOOP '89*, 1989.

24 Nathanael Scharli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP '03*, 2003.

25 Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 1995.

26 Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 1995.

27 Marc Van Limberghen and Tom Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

28 Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. Classless java. In *GPCE '16*, 2016.

29 Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety prooffor multiple inheritance in c++. In *OOPSLA '06*, 2006.

30 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

31 Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL '05*, 2005.