Hierarchical Dispatching and Overriding

Authors Omitted

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. One of the most promising approaches to multiple inheritance is the *trait* model. Traits offer a restricted model of multiple inheritance that is easy to reason and have many elegant properties. Traits have good support for method conflicts which arise from *diamond inheritance*: conflicts that arise from method implementations sharing a common ancestor. However, the mechanisms of traits 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 **FHJ** (short for ***F**eatherweight **H**ierarchical **J**ava*) that deals with unintentional method conflicts. In our new model, conflicting methods arising from unrelated methods can coexist in the same component. Moreover, they can be *independently overridden*. What ensures unambiguity is the use of information about the class hierarchy. In turn, hierarchical information is also employed in method dispatching, which uses a combination of static and dynamic type information to choose the implementation of a method at run-time. We give illustrative examples of our language and features and formalize **FHJ** as a minimal Featherweight-Java style calculus.

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 [21] and mixins [3] to offer a disciplined form of multiple inheritance. Java 8 offers a simple variant of traits with 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* [20,22] (also known as "fork-join inheritance" [20]). In the diamond problem, inheritance could allow 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 [13], and many others. Existing languages and research have taken care of this case extensively. Other issues

including how multiple inheritance deals with state, have also been discussed quite extensively [24,13,23].

In contrast with the diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [21]. That is conflicting methods that 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*".

Unintentional method conflicts are less common than the diamond problem, nonetheless, they have severe effects in practice if not handled properly. Unfortunately, such an issue has not received much formal study before. In practice, existing languages only provide limited support for it. 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. Intuitively using one method to fuse two unrelated features is unsatisfactory. Therefore we need a better solution to keep both features separately during inheritance, so as not to break *independent extensibility* [26].

Some other workarounds or approaches include delegation and renaming/exclusion in the trait model. Yet they still have various drawbacks as we will discuss in Section 2. Closest to our work are mechanisms available in C++ and C# that allow for two unintentionally conflicting methods to coexist in a class. Among them, C++ is a representative that accepts the triangle inheritance and resolves the ambiguity by *static dispatching*. However, C++ has limited support for virtual methods with unintentional conflicts, and it will often throw errors when composing them. This is again unsatisfactory because virtual methods are pervasive in OOP and used for code reuse and extensibility. A problem with the C++ approach is that programmers can only use either static or dynamic dispatching separately, but dealing with unintentional method conflicts seems to require a combination of both.

This paper proposes *hierarchical dispatch*: a novel approach to method dispatching, which combines static and dynamic information. Using hierarchical dispatch, 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. To present this

idea, we introduce a formalized model **FHJ** in Section 3 based on Featherweight Java [11], together with theorems and proofs for type soundness. In the model we also propose *hierarchical overriding*, where method overriding can be applied only to one branch of the class hierarchy. Our model can be viewed as a generalization of the simplified trait model by providing additional support to the triangle inheritance.

In summary, our contributions are:

- **Hierarchical dispatch algorithm:** an algorithm that integrates both the static type and dynamic type for method dispatch, and hence ensures unambiguity as well as extensibility.
- **Hierarchical overriding:** a novel notion that allows methods to override on individual branches of the class hierarchy.
- **FHJ:** a formalized model based on Featherweight Java, supporting the above features. Some theorems and their proofs are attached to confirm the type soundness of the model.
- **Prototype implementation**[1]**:** a simple interpreter implemented in Scala that follows our formalization rules, incorporating parsing, type-checking and semantic evaluation.

2 A Running Example: DrawableDeck

This section illustrates the features of our model for resolving unintentional method conflicts. As mentioned before, such a case arises when two inherited methods happen to have the same signature, but with different semantics and functionality. This situation is troublesome to programmers that use multiple inheritance. Below we illustrate with a running example called `DrawableDeck`, which models a drawable deck of cards. Note that we use a Java-like syntax throughout the paper, and all types are defined with the keyword "`interface`". The concept is closely related to Java 8 interfaces with default methods [2] and traits. For simplicity, an interface in our model has the following characteristics:

- It allows multiple inheritance.
- Every method represents a behaviour and requires a body for its implementation (like Java 8 default methods). There are no abstract methods.
- It can be directly instantiated by the `new` keyword.

Furthermore for simplicity, our examples and formalization do not deal with state at this stage.

2.1 Problem 1: 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

[1] Available in supplementary material attached to the submission.

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 illustration, the implementation of `draw` in `Drawable` only creates a blank canvas on the screen, and later some instances (`Circle`, `Square`, or others) that extend `Drawable` may define their own `drawObject` methods and invoke `draw`.

```
interface Deck {
  void draw() { // draws a card from the Deck
    Stack<Card> cards = this.getStack();
    if (!cards.isEmpty()) { Card card = cards.pop(); ... }
  }
}
interface Drawable {
  void draw() { // draws something on the screen
    JFrame frame = new JFrame("Canvas");
    frame.setVisible(true);
    ...
  }
}
```

Note that both methods have `void` return type (we will not formalize `void` in our calculus afterwards; here it is only for illustration). In `Deck`, `draw` tries to get the cards as a stack, pops out the top card, and so on. While in `Drawable`, `draw` creates a blank canvas using `JFrame`. Now, a programmer is designing a card game with GUI. He may want to draw a deck on the screen, so he defines a drawable deck using multiple inheritance:

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

The point of using multiple inheritance is surely for composing 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 throw a compile error on `DrawableDeck`, since the two `draw` methods cause a conflict, though accidentally.

Now one may quickly come up with a workaround, which is creating a new `draw` method in `DrawableDeck` to explicitly override the old ones. However, the two conflicting methods have totally different functionalities, merging them into one does not make any sense. This non-solution will hide the old methods and break independent extensibility. The feature we expect is called "triangle inheritance", that is to say, when the conflicting methods serve for different behaviours, we can allow them to coexist in the extending interface in a type-safe way.

Potential Fixes There are several workarounds that come to our mind:

- **I. Delegation.** As an alternative to multiple inheritance, delegation can be used by introducing two fields with `Drawable` type and `Deck` type, respectively. This avoids method conflicts. Nevertheless, 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 may not be possible if 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.
- **IV. Static dispatch.** Some other languages like C++ have direct support for the triangle inheritance. While conflicting methods are both preserved, the ambiguity of method call is resolved by static dispatch. Qualified names are used to specify the branch to give the right implementation.

Our solution: below is our code that shows the triangle inheritance. We can define `DrawableDeck` by inheriting `Drawable` and `Deck` together while type safety is ensured. Again it is slightly different from our prototype as we do not model `void` or statements; the code is just for illustration.

```
interface DrawableDeck extends Drawable, Deck {}
interface Foo {
  void func(Deck obj) { obj.draw(); }
}
// main program
DrawableDeck d = new DrawableDeck();
new DrawableDeck().Drawable::draw();  // calling draw in Drawable
new Foo().func(new DrawableDeck());   // calling draw in Deck
```

Note that both cases in the above code are valid in C++ for disambiguation. As shown by `func`, C++ looks at the static type of the receiver for static dispatching. Normally we can define such a method `func` for "implicit" casts in **FHJ**. C++ further allows explicit casts.

In the following text, we will see that C++ is still unsatisfactory, as it introduces another problem on code reuse, which motivates our work.

2.2 Problem 2: Dynamic Dispatch for Code Reuse

On the other hand, we also need dynamic dispatch as it is essential and widely used in Object-Oriented programming. C++ has the flexibility for choosing either way of dispatch by the `virtual` keyword. Unfortunately, this approach is still unsatisfactory regarding code reuse. For instance, we redefine `Deck` to support both `draw` and another operation called `shuffleAndDraw`:

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

`shuffleAndDraw` is a typical method that invokes `draw` in its definition. In principle, we want that invocation to use dynamic dispatch, because a programmer may define a subtype of `Deck` and override `draw`:

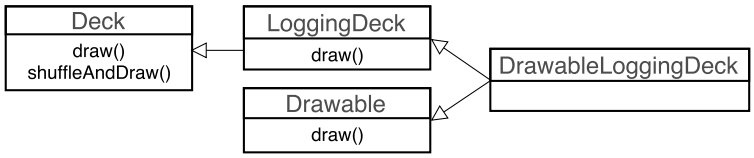


Fig. 1. UML graph for `DrawableLoggingDeck`.

```
interface LoggingDeck extends Deck {
  void draw() { // overriding
    Stack<Card> cards = this.getStack();
    if (!cards.isEmpty()) {
      Card card = cards.pop();
      println("The card is: " + card.toString());
      ...
    } else println("Empty deck.");
  }
}
```

In that case, `shuffleAndDraw` needs to be adapted to the new `draw`. But due to static dispatch, it still refers to the old `draw` in `Deck`. Thus programmers have to copy the `shuffleAndDraw` code in `LoggingDeck`. It requires a lot of redundant work if we have to duplicate all such methods. Hence in order for code reuse, we cannot just rely on static dispatch. However, as seen before, dynamic dispatch would take us back to problem 1, for instance, when we have the UML in Figure 1 and the following code:

```
interface DrawableLoggingDeck extends Drawable, LoggingDeck {...}
// main program
DrawableLoggingDeck d = new DrawableLoggingDeck();
d.shuffleAndDraw(); // ambiguous draw if dynamically dispatched
```

If we try to keep the triangle inheritance while applying dynamic dispatch, calling `shuffleAndDraw` would trigger ambiguity. In principle, we want `shuffleAndDraw` to invoke `LoggingDeck.draw` because they belong to the same branch, nonetheless, choosing either static dispatch or dynamic dispatch cannot ensure both the triangle inheritance and code reuse. As far as we know, C++ does not have a solution to it. Therefore we need to find another algorithm for method resolution.

Our solution: we propose *hierarchical dispatch* for method lookup. A qualified method invocation in our model, for instance, `e.I::m()`, is read as "finding the most specific method `m` along the path `I`". The meaning of "along the path `I`" is that, if the result of hierarchical dispatch is `J.m()` for some `J`, then such a `J` must be a super type of `e`'s dynamic type, and `J` has a subtyping relation with `I` (either `J <: I` or `J >: I`). Intuitively, the most specific `m()` must be from branch `I`, but it can be an updated version after `I` because of dynamic dispatch. The formal definition will be introduced later.

On the other hand, `e.m()` where `e` has static type `I`, simply behaves the same as `e.I::m()` in our model. Such a dispatch algorithm makes use of both the static type and the dynamic type of the receiver, so it is a combination of static and dynamic dispatch. Intuitively, the static type specifies one branch to avoid

ambiguity, and the dynamic type finds the latest version on that branch. It may still introduce ambiguity when there are multiple paths from the static type to the dynamic type, and those paths cause conflicts. We disallow this kind of conflict to ensure unambiguity. That is to say, we do not allow two methods to override a same base method when they cause conflicts. This is natural, as they are just two versions of the same operation, hence it is no longer an "unintentional" conflict, but the diamond problem.

Back to the example, we can write `"this.Deck::draw()"` inside the implementation of `shuffleAndDraw`, and on `d.shuffleAndDraw()` it automatically dispatches to `LoggingDeck.draw()`. Furthermore, the following code also behaves as expected:

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

It is because the compiler is able to know that the receiver "`this`" exactly has static type `Deck`. Hence hierarchical dispatch eliminates ambiguity.

2.3 Problem 3: Overriding on Individual Branches

We have seen that hierarchical dispatch can dynamically find the latest implementation of a method from one path (or branch). But when several branches are merged by triangle inheritance, there is usually demand for updating them separately. For example, someone plans to reuse the other features of `DrawableLoggingDeck`, but updates the `draw` from `Drawable` by setting the canvas invisible. Unfortunately in traditional models, the merged branches cannot be separately overridden any longer, since overriding will hide all branches and break coexistence. Modifying existing code is again unsatisfactory as it affects modularity.

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

```
interface DrawableLoggingDeck2 extends DrawableLoggingDeck {
  void draw() override Drawable {
    JFrame frame = new JFrame("Canvas");
    frame.setVisible(false);
    ...
  }
}
// main program
DrawableLoggingDeck2 d = new DrawableLoggingDeck2();
d.Drawable::draw()    // calling draw in DrawableLoggingDeck2
```

Terminology The `draw` methods we saw before this example are called "original methods" in this paper, because they are originally defined in its interface. In contrast, `DrawableLoggingDeck2` defines a "hierarchical overriding" method. The

difference is that traditional overriding overrides all branches by defining another original method, whereas hierarchical overriding only refines one branch.

In our model, 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. A quick counter-example is when there are two hierarchical overriding methods on `Drawable` in `DrawableLoggingDeck2`, it leads to ambiguity. The compiler is supposed to forbid that during compile time.

Two rules for hierarchical overriding: 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 `DrawableLoggingDeck2`, because existing two branches are `Drawable.draw` and `LoggingDeck.draw`, while `Deck.draw` is already covered. It does not really make sense to refine an old branch.

Similar to many OO languages, the model also allows *super method invocation* in a method body. The invocation `super.T::m()` will ignore all the subtypes of `T`, and only look at `T` together with its super interfaces. It should behave the same as `new T().m()` in principle.

3 Formalization

In this section, we present the formal model **FHJ** (***F**eatherweight **H**ierarchical **J**ava*) , based on Featherweight Java [11]. We use the same conventions as FJ; $\bar{\mathsf{I}}$ is shorthand for the (possibly empty) list $\mathsf{I}_1, ..., \mathsf{I}_n$, which may be indexed by I_i. The syntax, typing rules, and small-step semantics are presented below.

3.1 Syntax

Figure 2 shows the syntax of **FHJ**. The multiple inheritance feature of **FHJ** is inspired by Java 8 interfaces, which support 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 the methods declared in an interface are default methods, that is to say, they always provide default implementations. Object creation is directly supported via `new` I(). Fields and primitive types are not modelled in **FHJ**.

We use uppercase letters like $\mathsf{I}, \mathsf{J}, \mathsf{K}$ for interface identifiers. Due to multiple inheritance, an interface can have a set of super interfaces, where such a set can be empty. Each interface has a set of method declarations. Each method body returns an expression. As seen in Figure 2, we have introduced the `override` keyword to override an old implementation of the method. If the interface that it overrides is exactly the enclosing interface, then such a method is seen as "originally defined"; otherwise it is hierarchical overriding. For simplicity, overloading is not modelled for methods, which implies we can uniquely identify a method by its name.

Expressions can be standard constructs such as variables, method invocation, or object creation. A novel type of expressions are path-invocations like "$\mathsf{e}.\mathsf{I} :: \mathsf{m}(\bar{\mathsf{e}})$", meaning that the dynamically bound implementation for method m

$$
\begin{array}{lll}
\text{Interfaces} & \text{IL} & ::= \texttt{interface I extends } \overline{\text{I}}\ \{\overline{\text{M}}\} \\
\text{Methods} & \text{M} & ::= \text{I } m(\overline{\text{I}_x}\ \overline{x})\ \texttt{override J } \{\texttt{return } e;\} \\
\text{Expressions} & e & ::= x \mid e.m(\overline{e}) \mid \texttt{new I}() \mid e.\text{I} :: m(\overline{e}) \mid \texttt{super.I} :: m(e) \mid \langle \text{I} \rangle e \\
\text{Context} & \Gamma & ::= \overline{x} : \overline{\text{I}} \\
\text{Values} & v & ::= \langle \text{I} \rangle \texttt{new J}()
\end{array}
$$

Fig. 2. Syntax of **FHJ**.

should be along the path I. Another interesting expression are super-invocations, which enable a method to access an old implementation from the specified super type. Hence a super-invocation can only be used inside an interface definition. Finally, an expression can also be annotated with its static type. But *annotated expressions are only intended for the semantic rules*, hence they are not supposed to appear in the source program. A value "$\langle \text{I} \rangle \texttt{new J}()$" is the final result of multiple reduction steps evaluating an expression. Such value represents an object instance of J with an annotated static type I.

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

3.2 Subtyping and Typing Rules

The subtyping of **FHJ** consists of only a few rules shown at the top of Figure 3. In short, subtyping relations are built from the inheritance in interface declarations. Subtyping is both reflexive and transitive.

Details of type-checking rules are displayed at the bottom of Figure 3, 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 "this" type, namely the enclosing type. The three rules for method invocation, (T-Invk), (T-PathInvk) and (T-SuperInvk) are very similar, in the sense that they all check the type of the specific method, by using an auxiliary function `mtype`. `mtype` is the function for looking up method types, which we will illustrate later in Section 4. After the method type is obtained, they all check that arguments and the receiver have compatible types. Additionally, (T-PathInvk) requires the receiver to be the subtype of the specified path type, and (T-SuperInvk) checks if the enclosing type directly extends the specified super type.

The method typing rule (T-Method) is more interesting since the method can either be an original implementation or hierarchical overriding. Besides type-checking the return expression, we further use the helper function `mostSpecific`, again formally defined in Section 4. By the equation "$\texttt{mostSpecific}(m, \text{I}, \text{J}) = \{\text{J}\}$" we define the legality of method overrides as mentioned in Section 2.3, namely if it is hierarchical overriding, it should not step over original methods, otherwise their enclosing types would be returned instead of J.

Finally, (T-Invk) defines interface checking. The condition helps to ensure unambiguity and type soundness of the calculus. We will introduce `mbody` and some counter-examples later. But intuitively, if $\texttt{mbody}(m, \text{I}, \text{J})$ is defined, that

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

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

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

$$\text{(T-Invk) } \frac{\Gamma \vdash e_0 : I_0 \qquad \texttt{mtype}(m, I_0) = \overline{J} \to I \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J}}{\Gamma \vdash e_0.m(\overline{e}) : I}$$

$$\text{(T-PathInvk) } \frac{\Gamma \vdash e_0 : I_0 \qquad I_0 <: J_0 \qquad \texttt{mtype}(m, J_0) = \overline{J} \to I \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J}}{\Gamma \vdash e_0.J_0 :: m(\overline{e}) : I}$$

$$\text{(T-SuperInvk) } \frac{\texttt{ext}(\Gamma(\text{this}), J_0) \qquad \texttt{mtype}(m, J_0) = \overline{J} \to I \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J}}{\Gamma \vdash \texttt{super}.J_0 :: m(\overline{e}) : I}$$

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

$$\text{(T-Method) } \frac{\begin{array}{c} I <: J \qquad \texttt{mostSpecific}(m, I, J) = \{J\} \\ \texttt{mtype}(m, J) = \overline{I_x} \to I_e \qquad \overline{x} : \overline{I_x}, \text{this} : I \vdash e_0 : I_0 \qquad I_0 <: I_e \end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override } J\ \{\texttt{return } e_0;\}\ \text{OK IN } I}$$

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

Fig. 3. Typing and subtyping rules.

means `new` I().J :: m() is not ambiguous during runtime. Therefore the condition says that if an expression type checks, it should not introduce ambiguity during runtime in any case. The interface check is responsible for capturing ambiguity during compilation.

3.3 Small-step Semantics and Congruence

Figure 4 defines the small-step semantic and congruence rules, respectively. When evaluating an expression, they are invoked recursively and alternately to produce a single value in the end. The small-step semantics rules (S-Invk), (S-PathInvk) and (S-SuperInvk) behave similarly: each corresponds to one kind of method invocation. They all invoke `mbody` for method body lookup. Generally, one can understand $\texttt{mbody}(m, I, J)$ in a way that it finds the most specific body of method m, when the receiver has dynamic type I and static type J. Notice, for example in (S-Invk), that when "`new` I()" replaces "`this`", its static type should be the interface to which m is dispatched. Therefore we also keep the interface type

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

$$\text{(S-PathInvk)}\ \frac{\texttt{mbody}(m, I, K) = (I_0, \overline{I_x}\ \overline{x}, I'_e\ e_0)}{(\langle J\rangle \texttt{new}\ I())\,.K :: m(\overline{\langle I_e\rangle e}) \to \langle I'_e\rangle[\overline{\langle I_x\rangle e}/\overline{x}, \langle I_0\rangle \texttt{new}\ I()/\texttt{this}]e_0}$$

$$\text{(S-SuperInvk)}\ \frac{\texttt{mbody}(m, K, K) = (I_0, \overline{I_x}\ \overline{x}, I'_e\ e_0)}{\texttt{super}.K :: m(\overline{\langle I_e\rangle e}) \to \langle I'_e\rangle[\overline{\langle I_x\rangle e}/\overline{x}, \langle I_0\rangle \texttt{new}\ K()/\texttt{this}]e_0}$$

$$\text{(C-Receiver)}\ \frac{e_0 \to e'_0}{e_0.m(\overline{e}) \to e'_0.m(\overline{e})} \qquad \text{(C-PathReceiver)}\ \frac{e_0 \to e'_0}{e_0.K :: m(\overline{e}) \to e'_0.K :: m(\overline{e})}$$

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

$$\text{(C-PathArgs)}\ \frac{e \to e'}{e_0.I :: m(\ldots, e, \ldots) \to e_0.I :: m(\ldots, e', \ldots)}$$

$$\text{(C-SuperArgs)}\ \frac{e \to e'}{\texttt{super}.I :: m(\ldots, e, \ldots) \to \texttt{super}.I :: m(\ldots, e', \ldots)}$$

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

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

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

Fig. 4. Small-step semantics.

in the definition of `mbody`. This point is consistent with the last paragraph in Section 2.2.

On the other hand, there is a relationship between path invocation and the regular method invocation, and it can be observed from the similarity between their semantic rules in Figure 4. For any `e.I::m()`, the result of evaluation remains unchanged if we set the static type of `e` to be I. This can be done by an implicit cast, that is, we can define a function with one parameter type I, then `e` is passed to that function and directly returned. This is equivalent to writing explicit casts like `((I) e).m()` in languages like Java.

3.4 Properties

Previously the definitions of our model are given, now we should proceed to prove the type soundness of the model, which relates typing to computation. The type soundness states that, if an expression is well-typed, then after many

reduction steps it must reduce to a value, with its annotation to be a subtype of the original expression type. Following the Featherweight Java paper [11], the type-soundness theorem (Theorem 3) is proved by using the standard technique of subject reduction (Theorem 1) and progress (Theorem 2) [25]. In Theorem 2 "$\#(\overline{\mathsf{x}})$" denotes the number of elements.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash \mathsf{e} : \mathrm{I}$ *and* $\mathsf{e} \to \mathsf{e}'$, *then* $\Gamma \vdash \mathsf{e}' : \mathrm{I}'$ *for some* $\mathrm{I}' <: \mathrm{I}$.

Proof. See Appendix A.1. □

Theorem 2 (Progress). *Suppose* e *is a well-typed expression*

1. *If* e *includes* $(\langle \mathrm{J} \rangle \texttt{new}\ \mathrm{I}())\,.\mathsf{m}(\overline{\langle \mathrm{I_e} \rangle}\ \overline{\mathsf{e}})$ *as a subexpression, then* $\texttt{mbody}(\mathsf{m}, \mathrm{I}, \mathrm{J}) = (\mathrm{J}', \overline{\mathrm{I_x}}\ \overline{\mathsf{x}}, \mathrm{I}'_{\mathrm{E}}\ \mathsf{e}_0)$ *and* $\#(\overline{\mathsf{x}}) = \#(\overline{\mathsf{e}})$ *for some* $\overline{\mathrm{I_x}}, \overline{\mathsf{x}}, \mathrm{I}'_{\mathsf{e}}$ *and* e_0.
2. *If* e *includes* $(\langle \mathrm{J} \rangle \texttt{new}\ \mathrm{I}())\,.\mathrm{K} :: \mathsf{m}(\overline{\langle \mathrm{I_e} \rangle}\overline{\mathsf{e}})$ *as a subexpression, then* $\texttt{mbody}(\mathsf{m}, \mathrm{I}, \mathrm{K}) = (\mathrm{J}', \overline{\mathrm{I_x}}\ \overline{\mathsf{x}}, \mathrm{I}'_{\mathrm{E}}\ \mathsf{e}_0)$ *and* $\#(\overline{\mathsf{x}}) = \#(\overline{\mathsf{e}})$ *for some* $\overline{\mathrm{I_x}}, \overline{\mathsf{x}}, \mathrm{I}'_{\mathsf{e}}$ *and* e_0.
3. *If* e *includes* $\texttt{super}.\mathrm{K} :: \mathsf{m}(\overline{\langle \mathrm{I_e} \rangle}\overline{\mathsf{e}})$ *as a subexpression, then* $\texttt{mbody}(\mathsf{m}, \mathrm{K}, \mathrm{K}) = (\mathrm{J}', \overline{\mathrm{I_x}}\ \overline{\mathsf{x}}, \mathrm{I}'_{\mathrm{E}}\ \mathsf{e}_0)$ *and* $\#(\overline{\mathsf{x}}) = \#(\overline{\mathsf{e}})$ *for some* $\overline{\mathrm{I_x}}, \overline{\mathsf{x}}, \mathrm{I}'_{\mathsf{e}}$ *and* e_0.

Proof. See Appendix A.1. □

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

Proof. Immediate from Theorem 1 and Theroem 2. □

4 Auxiliary Definitions

In this section, we present the auxiliary definitions used in our formalization in Figure 5. To begin with, we introduce the basic functions: `ext`, `overrideSet` and `prune`. $\texttt{ext}(\mathrm{I}, \mathrm{J})$ simply indicates that interface I directly extends interface J. $\texttt{overrideSet}(\mathrm{I}, \mathrm{J})$ returns a set of methods defined in I that have "`override` J" in their signatures. Notice that $\texttt{overrideSet}(\mathrm{I}, \mathrm{I})$ is a special representative of the "originally-defined" method set from I. The `prune` function takes a set of types, and filters out those that have subtypes in the same set. Finally in the returned set, none of them has a subtyping to one another, since all super types have been removed.

4.1 `mostSpecific` and `mostSpecificOverride`

`mostSpecific` is an auxiliary function that finds the most specific original implementations of a method. Let us consider $\texttt{mostSpecific}(\mathsf{m}, \mathrm{I}, \mathrm{J})$, what it returns is a set of interfaces, each including its own m as a most specific implementation. Such a set may contain several elements, but that implies ambiguity; what we expect is actually a singleton set. By the definition of `mostSpecific` shown in Figure 5, an interface belongs to the return set if and only if:

$$\frac{\texttt{mostSpecific}(m, I_d, I_s) = \{I\} \qquad \texttt{mostSpecificOverride}(m, I_d, I) = \{J\} \qquad \texttt{interface J extends } \overline{J}\ \{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override I}\ \{\texttt{return } e_0;\}\ldots\}}{\texttt{mbody}(m, I_d, I_s) = (J, \overline{I_x}\ \overline{x}, I_e\ e_0)}$$

$$\frac{\begin{array}{c} set1 = \{K <: J \text{ and } K >: I \mid m \in \texttt{overrideSet}(K, K)\} \\ set2 = \{K >: J \mid m \in \texttt{overrideSet}(K, K)\} \end{array}}{\texttt{mostSpecific}(m, I, J) = \begin{cases} prune(set1) \text{ if } set1 \text{ is not empty} \\ prune(set2) \text{ otherwise} \end{cases}}$$

$$\frac{set = \{K <: J \text{ and } K >: I \mid m \in \texttt{overrideSet}(K, J)\}}{\texttt{mostSpecificOverride}(m, I, J) = prune(set)}$$

$$prune(set) = \{I \in set \mid \nexists J \in set, J <: I, J \neq I\} \qquad \frac{\texttt{interface I extends } \overline{I}\ \{\overline{M}\} \qquad J \in \overline{I}}{\texttt{ext}(I, J)}$$

$$\frac{\texttt{interface I extends } \overline{I}\ \{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override J}\ldots\}}{m \in \texttt{overrideSet}(I, J)}$$

Fig. 5. Auxiliary functions.

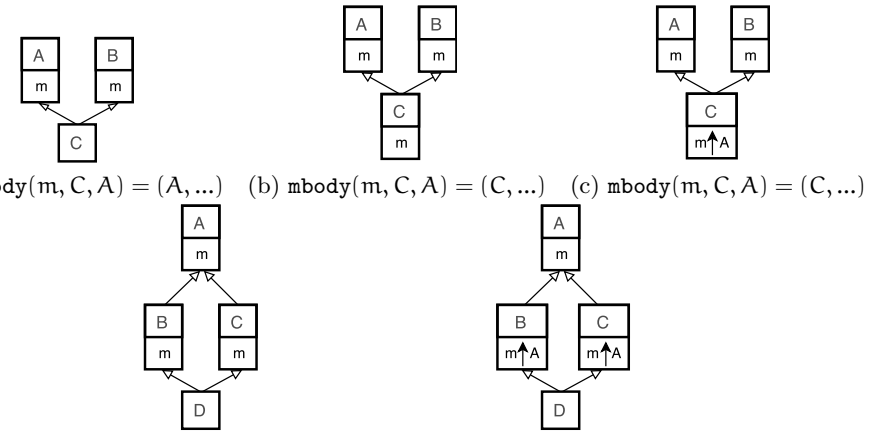
- It has an original definition of m;
- It is a supertype of I;
- It is along path J, meaning that it is either a supertype or a subtype of J (including J itself);
- It does not have a subtype in the same set because we have used $prune$ to filter out all super types.

We could have put $set1$ and $set2$ together, but the current definition is clearer.

The `mostSpecific` function only focuses on original method implementations, all the hierarchical overriding methods are omitted during that time. On the other hand, another auxiliary function $\texttt{mostSpecificOverride}(m, I, J)$ has the assumption that J defines an original m, and this function tries to find the most specific implementations that hierarchically overrides such an m. Just as `mostSpecific`, `mostSpecificOverride` also returns the set of interfaces after pruning. An interface belongs to the return set if and only if:

- It is between I and J;
- It hierarchically overrides J.m;
- It does not have a subtype in the same set.

The algorithm for finding the most specific hierarchical overriding method is quite similar to that for finding the most specific original method. A hierarchical overriding is not allowed to work on another hierarchical overriding, and one can hide another if their interfaces have subtyping relations. If they do not hide each other, the result implies ambiguity.



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

(d) $\texttt{mbody}(\mathtt{m}, \mathsf{D}, \mathsf{A}) = \texttt{undefined}$ (e) $\texttt{mbody}(\mathtt{m}, \mathsf{D}, \mathsf{A}) = \texttt{undefined}$

Fig. 6. Examples on `mbody`. "↑" stands for hierarchical overriding.

4.2 `mbody` and `mtype`

$\texttt{mbody}(\mathtt{m}, \mathrm{I_d}, \mathrm{I_s})$, as defined in Figure 5, denotes a method body lookup function. We use $\mathrm{I_d}, \mathrm{I_s}$, since `mbody` is usually invoked by a receiver of a method $\mathtt{m}$, with its dynamic type $\mathrm{I_d}$ and static type $\mathrm{I_s}$. Such a function returns the most specific method implementation, more accurately, its parameters, returned expression and the types. It considers both originally defined methods and hierarchical overriding methods, so `mostSpecific` and `mostSpecificOverride` are invoked.

To calculate $\texttt{mbody}(\mathtt{m}, \mathrm{I_d}, \mathrm{I_s})$:

- First, it invokes $\texttt{mostSpecific}(\mathtt{m}, \mathrm{I_d}, \mathrm{I_s})$ and returns a set.
- If `mostSpecific` returns a singleton set $\{\mathrm{I}\}$, then it is good, otherwise, `mbody` is undefined in this case. The set $\{\mathrm{I}\}$ implies that we will use the $\mathtt{m}$ from I without ambiguity. Moreover, we have to invoke $\texttt{mostSpecificOverride}(\mathtt{m}, \mathrm{I_d}, \mathrm{I})$, to check if there are updated versions of $\mathtt{m}$ between $\mathrm{I_d}$ and I. Again we forbid ambiguity, so the expected set after pruning is also a singleton set $\{\mathrm{J}\}$.
- Finally, we fetch the implementation of $\mathtt{m}$ in interface J and return its related information.

The definition of `mtype` used in typing rules simply relies on `mbody`. In short,

$$\texttt{mbody}(\mathtt{m}, \mathrm{I}, \mathrm{I}) = (\mathrm{J}, \overline{\mathrm{I_x}}\ \overline{\mathrm{x}}, \mathrm{I_e}\ e) \implies \texttt{mtype}(\mathtt{m}, \mathrm{I}) = \overline{\mathrm{I_x}} \to \mathrm{I_e}$$

4.3 Examples

Examples for `mbody` are shown in Figure 6. Note that we use `m` to denote an original method, and "`m`↑`A`" for hierarchical overriding on `A`. For each small example, the result gives the interface to which $\mathtt{m}$ is dispatched. (a) is a basic model for unintentional method conflicts; (b) and (c) demonstrate that hierarchical dispatch

can find the most specific original method and hierarchical overriding method. More interesting are the two bad examples (d) and (e), they both fail on `mbody`. (d) is the well-known diamond inheritance, which our model also forbids, and (e) is similar to (d) because the two hierarchical overriding methods are working on the same operation. Both counter-examples imply that `new D().A::m()` will lead to ambiguity, and in order for type soundness, both have to be denied by the type checker. This is guaranteed by the interface check (T-Intf) in Figure 3.

5 Related Work

Here we describe related work in four parts. We discuss from mainstream popular multiple inheritance models to several specific models (C++ and C#) which are closest to our work. Then we discuss similar techniques used in **SELF**. Finally, we discuss the foundation and related work of our formalization.

5.1 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 [20,22]). There are many existing languages/models that support multiple inheritance [8,17,3,21,13,14,15,9,1]. The Mixin model [3] allows 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 [17] are in fact linearized mixins and hence have the same problem as mixins.

Simplifying the mixins approach, traits [21,7] draw a strong line between units of reuse and object factories. Traits act as units of reuse, containing functionality code; while classes, assembled from traits, act as object factories. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the `default` keyword) inside interfaces and there are also proposals such as FeatherTrait Java [12] for extending Java with traits. There are extensions [18,19] to the original trait model with various advanced features such as *renaming*. As discussed in Section 2, the renaming feature helps to solve the unintentional method conflicts problem, however, it breaks structural subtyping.

Malayeri and Aldrich proposed a model CZ [13] 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, and they handle method conflicts in the same way, by simply disallowing two methods with the same signature from two different units to coexist. The ambiguity is

resolved either by programmers explicitly or by setting a linearised composition. This is the common drawback of the above-mentioned models versus our model.

5.2 Resolving Unintentional Method Conflicts

There are a few languages that partly realize the problem of unintentional conflicts and provide some support for it.

C++ model. C++ supports a very flexible inheritance model and allows programmers to choose either static dispatch or dynamic dispatch for method lookup. It allows unintentional conflicts and uses static dispatch to resolve them, as discussed in Section 2. With virtual methods, dynamic dispatch is used and method lookup algorithm will find the most specific method definition. Although C++ support this flexibility, dynamic dispatching on unintentional conflicting methods is problematic. Furthermore, C++ does not support hierarchical overriding compared to our model.

C# Explicit method implementation. Explicit method implementation is a special feature supported by C#. As described in C# documentation [14], a class that implements an interface can explicitly implement a member of that interface. When a member is explicitly implemented, it can only be accessed through an instance of the interface. Explicit interface implementation allows the programmer to inherit two 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 implementations are not allowed in C# interfaces; (2) there is only one level of hierarchical overriding to refine conflicting methods individually.

5.3 Hierarchical Dispatch

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. It does not include classes but instead allow 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. It specifies that "*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*". Similar to our model, this sender path tiebreaker rule resolves ambiguities between unrelated

abstractions. However, it is used in a prototype-based language setting and it does not support method hierarchical overriding as **FHJ** does.

5.4 Formalization Based on Featherweight Java

Featherweight Java (FJ) [11] is a minimal core calculus of Java language, proposed by Igarashi et. al. There are many models built on Featherweight Java, including FeatherTrait [12], Featherweight defenders [10], Jx [16], Featherweight Scala [6], and so on. FJ provides the standard model of formalizing a Java-like object-oriented language and is easily extensible. In terms of formalization, the key novelty of our model is the use of static types as annotations along with various terms. As far as we know, this technique has not appeared in the literature before. The static type annotation is of vital importance in our hierarchical dispatch algorithm.

6 Conclusion

This paper proposes **FHJ** as a new multiple inheritance model for unintentional method conflicts. Existing approaches including static dispatch and dynamic dispatch have to compromise between code reuse and type safety, whereas **FHJ** uses a different solution called hierarchical dispatch to obtain both. Such an approach not only offers great code reuse like dynamic dispatch but also ensures unambiguity by our algorithm for method resolution. This paper also introduces hierarchical overriding that refines branches individually. **FHJ** is formalized with a basis on Featherweight Java and supported by a few theorems. The prototype is implemented in Scala as a simple interpreter.

Our model can certainly be improved at some aspects. We did not formalize fields as in Featherweight Java for simplicity. And we did not formalize other common features including method overloading, casts, covariant return types, and so on, some of which are orthogonal in the design space. Moreover, we restrict that hierarchical overriding can only work on original methods. Potentially a looser condition can better support encapsulation and modularity with respect to code design.

References

1. Ancona, D., Lagorio, G., Zucca, E.: Jam—designing a java extension with mixins. ACM Trans. Program. Lang. Syst. 25(5), 641–712 (2003)
2. Bono, V., Mensa, E., Naddeo, M.: Trait-oriented programming in java 8. In: PPPJ'14 (2014)
3. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA/ECOOP '90 (1990)
4. Chambers, C., Ungar, D., Chang, B.W., Hölzle, U.: Parents are shared parts of objects: Inheritance and encapsulation in self. Lisp Symb. Comput. 4(3) (Jul 1991)
5. COOK, S.: Varieties of inheritance. In: OOPSLA'87 Addendum To The Proceedings. OOPSLA'87 Panel P2 (1987)

6. Cremet, V., Garillot, F., Lenglet, S., Odersky, M.: A core calculus for scala type checking. In: Proceedings of the 31st International Conference on Mathematical Foundations of Computer Science. MFCS'06 (2006)
7. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. ACM Trans. Program. Lang. Syst. 28(2), 331–388 (2006)
8. Ellis, M.A., Stroustrup, B.: The annotated C++ reference manual. Addison-Wesley (1990)
9. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'98 (1998)
10. Goetz, B., Field, R.: Featherweight defenders: A formal model for virtual extension methods in java. `http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf` (2012)
11. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: A minimal core calculus for java and gj. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (2001)
12. Liquori, L., Spiwack, A.: Feathertrait: A modest extension of featherweight java. ACM Trans. Program. Lang. Syst. 30(2), 11 (2008)
13. Malayeri, D., Aldrich, J.: Cz: Multiple inheritance without diamonds. In: OOPSLA '09 (2009)
14. Microsoft: Csharp explicit interface member implementations document. `https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx` (2003)
15. Moon, D.A.: Object-oriented programming with flavors. In: Conference Proceedings on Object-oriented Programming Systems, Languages and Applications. OOPSLA'86 (1986)
16. Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. OOPSLA '04 (2004)
17. Odersky, M., al.: An overview of the scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland (2004)
18. Reppy, J., Turon, A.: A foundation for trait-based metaprogramming. In: International workshop on foundations and developments of object-oriented languages (2006)
19. Reppy, J., Turon, A.: Metaprogramming with traits. In: Proceedings of the 21st European Conference on Object-Oriented Programming. ECOOP'07 (2007)
20. Sakkinen, M.: Disciplined inheritance. In: ECOOP'89 (1989)
21. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP'03 (2003)
22. Singh, G.B.: Single versus multiple inheritance in object oriented programming. SIGPLAN OOPS Mess. (1995)
23. Stroustrup, B.: The C++ programming language. Pearson Education India (1995)
24. Wang, Y., Zhang, H., Oliveira, B.C.d.S., Servetto, M.: Classless java. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. GPCE'16 (2016)
25. Wright, A., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)
26. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. In: FOOL'05 (2005)

A Appendix

A.1 Proofs

Lemma 1. *If* $\mathit{mbody}(m, I, J) = (J', \overline{I_x}\ \overline{x}, I_e\ e)$, *then for some* J_0 *with* $I <: J_0$, $\exists I' <: I_e$, *such that* $\overline{x} : \overline{I_x}, \mathit{this} : J_0 \vdash e : I'$.

Proof.
The base case: if m *is defined in* I, *then it is easy since* m *is defined in* I *and* $\overline{x} : \overline{I_x}, \texttt{this} : I \vdash e : I_e$, *by the rule* (T-Method). *The induction step is also straightforward.* □

Lemma 2 (Method Type Preservation). *If* $\mathit{mtype}(m, I) = \overline{I_x} \to I_e$, *then* $\mathit{mtype}(m, J) = \overline{I_x} \to I_e$ *for all* $J <: I$.

Proof.
Straight induction on the derivation of $J <: I$. Whether m is defined in J or not, $\texttt{mtype}(m, J)$ should be the same as $\texttt{mtype}(m, K)$ where J `extends` $K\{...\}$. □

Lemma 3 (Term Substitution Preserves Typing). *If* $\Gamma, \overline{x} : \overline{I_x} \vdash e : I$ *, and* $\Gamma \vdash \overline{y} : \overline{I_y}$ *where* $\overline{I_y} <: \overline{I_x}$, *then* $\Gamma \vdash [\overline{y}/\overline{x}]e : I'$ *for some* $I' <: I$.

Proof.
Case Var. $e = x \quad I = \Gamma(x)$.
If $x \notin \overline{x}$, then the conclusion is immediate, since $[\overline{y}/\overline{x}]x = x$. On the other hand, if $x = x_i$ and $I = I_{xi}$, then since $[\overline{y}/\overline{x}]x = [\overline{y}/\overline{x}]x_i = y_i$, letting $I' = I_{y_i}$ finishes the case.
Case New. $e = \texttt{new}\ I()$ and there are no term for substitution, the conclusion is obvious.
Case Invk. $e = e_0.m(\overline{e}) \quad \Gamma, \overline{x} : \overline{I_x} \vdash e_0 : I_0$

$$\mathit{mtype}(m, I_0) = \overline{I_e} \to J$$

$$\Gamma, \overline{x} : \overline{I_x} \vdash \overline{e} : \overline{I} \quad \overline{I} <: \overline{I_e}$$

By induction hypothesis, there are some I_0' and $\overline{I_e'}$ such that

$$\Gamma \vdash [\overline{y}/\overline{x}]e_0 : I_0' \quad I_0' <: I_0$$

$$\Gamma \vdash [\overline{y}/\overline{x}]\overline{e} : \overline{I_e'} \quad \overline{I_e'} <: \overline{I}$$

By lemma 2, $\mathit{mtype}(m, I_0') = \overline{I_e} \to J$, then $\overline{I_e'} <: \overline{I_e}$ by the transitivity of $<:$. Therefore, by the rule (T-Invk), $\Gamma \vdash [\overline{y}/\overline{x}]e_0.m([\overline{y}/\overline{x}]\overline{e}) : J$.
Case PathInvk. $e = e_0.I :: m(\overline{e})$ and proof is similar as case Var.
Case SuperInvk. $e = \texttt{super}.I :: m(\overline{e})$
Suppose $\Gamma \vdash \texttt{this} : I_0$, the following proof should be similar as case Var. □

Proof for Theorem 1

Proof.
Case Invk. let

$$e = \langle \mathsf{J} \rangle \texttt{new } \mathsf{I}().\mathfrak{m}(\overline{\langle \mathsf{I}_e \rangle \overline{e}})$$

Suppose

$$\texttt{mbody}(\mathfrak{m}, \mathsf{I}, \mathsf{J}) = (\mathsf{J}', \overline{\mathsf{I}_x}\ \overline{x}, \mathsf{I}_{e_0}\ e_0)$$

then

$$e' = [\overline{\langle \mathsf{I}_x \rangle \overline{e}}/\overline{x},\ \langle \mathsf{J} \rangle \texttt{new } \mathsf{I}()/\texttt{this}]e_0$$

By rules (T-New) and (T-Invk),

$$\Gamma \vdash \texttt{new } \mathsf{I}() : \mathsf{I} \quad \texttt{mtype}(\mathfrak{m}, \mathsf{I}, \mathsf{J}) = \overline{\mathsf{I}_x} \to \mathsf{I}_{e_0} \quad \Gamma \vdash \overline{e} : \overline{\mathsf{I}'_e} \quad \overline{\mathsf{I}'_e} <: \overline{\mathsf{I}_x} \quad \textit{, for some } \overline{\mathsf{I}'_e}$$

By Lemma 1,

$$\Gamma, \overline{x} : \overline{\mathsf{I}_x}, \texttt{this} : \mathsf{J}_0 \vdash e_0 : \mathsf{I}_f \textit{, for some } \mathsf{J} <: \mathsf{J}_0 \textit{ and } \mathsf{I}_f <: \mathsf{I}_{e_0}$$

By Lemma 3,

$$\Gamma \vdash [\overline{\langle \mathsf{I}_x \rangle \overline{e}}/\overline{x},\ \langle \mathsf{J} \rangle \texttt{new } \mathsf{I}()/\texttt{this}]e_0 : \mathsf{I}_g \textit{, for some } \mathsf{I}_g <: \mathsf{I}_f$$

So $\mathsf{I}_g <: \mathsf{I}_{e_0}$, finally just let $\mathsf{I}' = \mathsf{I}_g$.
Case PathInvk. let

$$e = \langle \mathsf{J} \rangle \texttt{new } \mathsf{I}().\mathsf{K} :: \mathfrak{m}(\overline{\langle \mathsf{I}_e \rangle \overline{e}})$$

Suppose

$$\texttt{mbody}(\mathfrak{m}, \mathsf{I}, \mathsf{K}) = (\mathsf{J}', \overline{\mathsf{I}_x}\ \overline{x}, \mathsf{I}_{e_0}\ e_0)$$

then

$$e' = [\overline{\langle \mathsf{I}_x \rangle \overline{e}}/\overline{x},\ \langle \mathsf{K} \rangle \texttt{new } \mathsf{I}()/\texttt{this}]e_0$$

By rules (T-New) and (T-Invk),

$$\Gamma \vdash \texttt{new } \mathsf{I}() : \mathsf{I} \quad \texttt{mtype}(\mathfrak{m}, \mathsf{I}, \mathsf{K}) = \overline{\mathsf{I}_x} \to \mathsf{I}_{e_0} \quad \Gamma \vdash \overline{e} : \overline{\mathsf{I}'_e} \quad \overline{\mathsf{I}'_e} <: \overline{\mathsf{I}_x} \quad \textit{, for some } \overline{\mathsf{I}'_e}$$

By Lemma 1,

$$\Gamma, \overline{x} : \overline{\mathsf{I}_x}, \texttt{this} : \mathsf{J}_0 \vdash e_0 : \mathsf{I}_f \textit{, for some } \mathsf{K} <: \mathsf{J}_0 \textit{ and } \mathsf{I}_f <: \mathsf{I}_{e_0}$$

By Lemma 3,

$$\Gamma \vdash [\overline{\langle \mathsf{I}_x \rangle \overline{e}}/\overline{x},\ \langle \mathsf{K} \rangle \texttt{new } \mathsf{I}()/\texttt{this}]e_0 : \mathsf{I}_g \textit{, for some } \mathsf{I}_g <: \mathsf{I}_f$$

So $\mathsf{I}_g <: \mathsf{I}_{e_0}$, finally just let $\mathsf{I}' = \mathsf{I}_g$.
Case Super-Invk. let

$$e = \texttt{super}.\mathsf{K} :: \mathfrak{m}(\overline{\langle \mathsf{I}_e \rangle \overline{e}})$$

Suppose

$$\texttt{mbody}(\mathfrak{m}, \mathsf{K}, \mathsf{K}) = (\mathsf{J}', \overline{\mathsf{I}_x}\ \overline{x}, \mathsf{I}_{e_0}\ e_0)$$

then

$$e' = [\overline{\langle I_x \rangle} \overline{e} / \overline{x}] e_0$$

By rules (T-New) and (T-Invk),

$$\texttt{mtype}(m, K, K) = \overline{I_x} \to I_{e_0} \quad \Gamma \vdash \overline{e} : \overline{I'_e} \quad \overline{I'_e} <: \overline{I_x} \quad \textit{, for some } \overline{I'_e}$$

By Lemma 1,

$$\Gamma, \overline{x} : \overline{I_x}, \texttt{this} : J_0 \vdash e_0 : I_f \textit{, for some } K <: J_0 \textit{ and } I_f <: I_{e_0}$$

By Lemma 3,

$$\Gamma \vdash [\overline{\langle I_x \rangle} \overline{e} / \overline{x}] e_0 : I_g \textit{, for some } I_g <: I_f$$

So $I_g <: I_{e_0}$, finally just let $I' = I_g$.

□

Proof for Theorem 2

Proof.
Case 1. Given that e is well-typed, by rule (T-Invk), $\texttt{mtype}(m, J)$ is defined. By rule (T-Intf) for interface I,

$$\forall J >: I, m, \texttt{mtype}(m, J) \text{ is defined } \Rightarrow \texttt{mbody}(m, I, J) \text{ is defined}$$

So straightforwardly, we get $\texttt{mbody}(m, I, J) = (J', \overline{I_x}\ \overline{x}, I'_E\ e_0)$ and $\#(\overline{x}) = \#(\overline{e})$ for some $J', \overline{I_x}, \overline{x}, I'_e$ and e_0
Case 2 and 3. Proof for case 2 and 3 is similar to case 1, and we omit it here. □