NoTitle

Author 1, Author 2

Lab, University, Address

Abstract. Text.

1 Introduction

– problem: Unintended method confliction in Multiple inheritance
– Existing approaches or models and Their drawbacks
– New features (update)
– Our contributions

In multiple inheritance, naming conflicts often occurs. Among these conflicts, some are real conflicts which needs explicit resolve by programmers, however, there are cases where accidental naming conflicts occurs, where the conflicting methods have completely different meaning/domain which just share the same name.

Existing OOP models have taken care of the first case intensively. However, few of them supports unintended method confliction well. Trait and other mainstream OO models do not allow unintended methods confliction to co-exist. SELF [] uses *sender path tiebreaker rule* to automatically resolve ambiguities that are almost certainly caused by accidental naming conflicts. C++ allows methods with the same signature co-exist in a class via inheritance and programmers can use :: operator to select the method wanted. However none of them allows refining these unintended conflicting methods in subclasses. We propose a calculus that deals with unintended method confliction and meanwhile allows refining these methods.

Contributions:

– A multiple inheritance model formalized as **MIM**.
– Novel notion **updates**for method path updating.
– Implementation of a simple typechecker and evaluator in Scala.

2 A Running Example: DrawableDeck

H: with UML graphs the illustration could be better. This section illustrates the features of our **MIM** 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 functionalities. This could be quite troublesome to programmers that use multiple inheritance.

Below we illustrate with a running example called `DrawableDeck`. Note that we use Java-like syntax throughout the paper, and all types are defined with the keyword "**`interface`**", which supports multiple inheritance. Since **MIM** is designed to be simple, we do not allow abstract methods, that is every method is required to have a body for its implementation. In that case, interfaces can be directly instantiated by the keyword "**`new`**" to create an object.

2.1 Problem: Unintentional Method Conflicts

Suppose that two components `Drawable` and `Deck` have been developed in a system. `Drawable` defines an interface for graphics that can be drawn, which includes a method called `draw()` for visual display. While interface `Deck` represents a deck of cards, and supports several operations, like `draw()` for drawing a card from the deck.

```
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 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 of components, achieving great code reuse. It is supported by many mainstream OO languages. Nevertheless at this point, `DrawableDeck` has to throw a compile error, for the two `draw()` methods cause a conflict, though accidentally.

2.2 Potential Fixes

For that problem, there are several workarounds that quickly 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, delegation itself is too restricted in modularity, and meanwhile introduces a lot of boilerplate.

II. Creating a `draw()` method in `DrawableDeck`, which explicitly overrides the old ones. This is a non-solution. It does not make any sense to override both methods with totally different functionalities, as old methods have to be hidden.

III. Choosing one of them as the default method, like Mixins. The mixin model can be applied to choose a default one based on linearisation. Similarly, we want to preserve both features, rather than keeping only one of them.

IV. Method exclusion like traits. Same reason as above.

V. Method renaming like traits. This is probably what people do in most cases, by simply renaming one to avoid conflicts. It can indeed preserve both features, however, it is cumbersome in practice, as introducing new names can affect other code blocks. Certainly this is a workaround, not a solution.

What we really expect from the language is we keep both methods without renaming, and the type checker does not complain on the inheritance. But we need to find an approach to disambiguate on method calls statically.

Certainly the compiler can ignore the conflict when `DrawableDeck` is declared, but once an object of `DrawableDeck` is created, a method call for `draw()` on that object is ambiguous, due to dynamic dispatch. Nonetheless, we can adopt static dispatch for disambiguating. Some languages like C++ use qualified names in that way:

```
void func(Drawable obj) {
  obj.draw();
}

DrawableDeck d = new DrawableDeck();
d.Drawable::draw();     // calling draw() in Drawable
((Drawable) d).draw(); // calling draw() in Drawable
func(d);               // calling draw() in Drawable
```

Thus we have:

VI. Static dispatch. Static dispatch finds out and invokes the most specific method "by need".

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, here we redefine `Deck` to support both `draw()` and another operation called `shuffleAndDraw()`:

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

`shuffleAndDraw()` is a representative 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()`:

```
interface LoggingDeck {
  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.");
    }
  }
}
```

Usually we want to reuse the code of `shuffleAndDraw()` during inheritance, hence dynamic dispatch is necessary, otherwise programmers have to override all the other methods that invoke `draw()`. However, as seen before, dynamic dispatch can cause ambiguity if we have:

```
interface DrawableLoggingDeck extends Drawable, LoggingDeck {
  ...
}

DrawableLoggingDeck d = new DrawableLoggingDeck();
d.shuffleAndDraw(); // ambiguous draw()
```

Since the dynamic type of the receiver is `DrawableLoggingDeck`, calling `shuffleAndDraw ()` triggers the ambiguity. When `shuffleAndDraw()` invokes `draw()`, what we really want is `LoggingDeck.draw()`, yet neither static dispatch nor dynamic dispatch in languages like C++ does so. Therefore, we need to find another algorithm for method binding.

2.3 Solution in MIM: Hierarchical Dispatch

Our **MIM** model uses *hierarchical dispatch* for method lookup. A qualified method invocation, for instance, `e.I::m()`, is read as "finding the most specific `m ()` along path `I`". The meaning of "along 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 overridden version after `I` like dynamic dispatch. The formal definition will be introduced later.

On the other hand, `((I) e).m()`, or simply `e.m()` where `e` has static type `I`, behaves the same as `e.I::m()` in our model. Such a dispatch make uses of both the static type and the dynamic type of the receiver. Intuitively, the static type specifies one branch of the method 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. To prevent this, 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 in that way.

With the old example, below code meets our expectation:

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

It guarantees that `shuffleAndDraw()` are calling `shuffle()` and `draw()` from its own branch, so that the namesakes from other branches will not cause conflicts. Now `d.shuffleAndDraw()` is no longer ambiguous.

Actually in **MIM**, we can still write "`shuffle();`" and "`draw();`", because the compiler is able to know that the receiver "**`this`**" exactly has static type `Deck`, hence hierarchical dispatch eliminates ambiguity.

2.4 Method Update in MIM

MIM supports two kinds of polymorphism on methods: (1) method overriding; (2) method update. A method update is similar to overriding, in the sense that it provides a refined implementation of a method. But the difference is that an update just refines one branch, since **MIM** can preserve several conflicted branches at the same time. Whereas a method overriding will hide all its inherited original methods, in which case method conflicts are unexpectedly ignored.

For example, we define an interface `DrawableLoggingDeck2`, where we refine the `draw()` from `Drawable()` by setting the canvas invisible:

```
interface DrawableLoggingDeck2 extends DrawableLoggingDeck {
  void draw() updates Drawable {
    JFrame frame = new JFrame("Canvas");
    frame.setVisible(false);
    ...
  }
}
```

Here the update specifies `Drawable` to be refined. The use of this update ensures that both branches of `draw()` are preserved. At the same time, we should notice that a method update is no longer considered as an original method. Some client code is shown below:

```
DrawableLoggingDeck2 d2 = new DrawableLoggingDeck2();
d2.Drawable::draw(); // calling draw() in DrawableLoggingDeck2
d2.Deck::draw();     // calling draw() in LoggingDeck
```

In **MIM**, hierarchical dispatch first finds the most specific original method (with respect to method overriding), then it finds the most specific update on that method. A counter-example is when there are two method updates on `Drawable` in `DrawableLoggingDeck2`, it leads to ambiguity. The compiler is supposed to forbid that during compile time.

Two rules for method updates in **MIM**: updates can only refine **original** methods, and they cannot overlap method overriding. For instance, writing "`void draw() updates Deck {...}`" is disallowed in `DrawableLoggingDeck2`, because existing two branches are `Drawable.draw()` and `LoggingDeck.draw()`, yet `Deck.draw()` is already overridden. It does not really make sense to update the old branch.

Similar to many OO languages, **MIM** 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.

3 Formalization

In this section, we present a formalization of our MIM calculus, based on a minimal subset of Java 8 interfaces. The syntax, typing rules and small-step semantics are included below.

3.1 Syntax

Figure 1 shows the syntax of MIM. The multiple inheritance feature of MIM has a basis on Java interfaces. To demonstrate how unintentional method conflicts are untangled in MIM, we present the calculus in a straightforward way, hence 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. From this point we can view that we are actually modelling a class model that supports multiple inheritance. Then it is straightforward to do object creation like `new` $\mathsf{I}()$. Fields and primitive types are not modelled as well.

We use uppercase letters like $\mathsf{I}, \mathsf{J}, \mathsf{K}$ to represent identifiers for interfaces. By multiple inheritance, an interface can have a set of super interfaces, where such a set can be empty. Inside an interface are a set of method declarations. Each method body holds a return statement. As seen in Figure 1, 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”. Again for simplicity, overloading is not modelled for methods, which implies we can uniquely identify a method by its name.

An expression can be a variable, a method invocation, an object creation, furthermore, a path-invocation like "$e.\mathrm{I} :: m(\overline{e})$", meaning that the dynamically binded implementation for method m should be along the path I. Another case is the super-invocation, enabling 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 a value "$< \mathrm{I} >$ `new` $\mathrm{J}()$". It is exactly an object instance of J with annotated static type I. Note that values are only intended for the small-step semantics of MIM, hence they are not supposed to appear in the source program.

Interfaces	IL	$::= \texttt{interface}\ \mathrm{I}\ \texttt{extends}\ \overline{\mathrm{I}}\ \{\overline{M}\}$
Methods	M	$::= \mathrm{I}\ m(\overline{\mathrm{I}_x\ x})\ \texttt{override}\ \mathrm{J}\ \{\texttt{return}\ e;\}$
Expressions	e	$::= x \mid e.m(\overline{e}) \mid \texttt{new}\ \mathrm{I}() \mid e.\mathrm{I} :: m(\overline{e}) \mid \texttt{super}.\mathrm{I} :: m(e) \mid v^*$
Context	Γ	$::= \overline{x : \mathrm{I}}$
Values	v	$::= < \mathrm{I} > \texttt{new}\ \mathrm{J}()$

Interface names	$\mathrm{I}, \mathrm{J}, K$
Method names	m
Variable names	x

Fig. 1. Syntax. *: only intended for semantic rules. Do we need "this"?

3.2 Subtyping and Typing Rules

The subtyping of MIM consists of only a few rules shown in Figure ?. In short, subtyping relations are built from the inheritance in interface declarations. They hold both reflexivity and transitivity.

Details of type-checking rules are displayed in Figure ?, 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. 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 a update. Besides static type-checking for the return expression, we further use the helper function `mostSpecific` to ensure that the method update is legal. The formal definition is available in

Section ? By that we define the legality of method updating: the updated method must be an original method (not another method update), and method updating cannot cross over method overriding (a method overriding cannot appear between the method update and the updated original method in inheritance hierarchy).

Finally, (T-INTF) defines interface type-checking in a straightforward way. Note that it is responsible for checking conflicted paths with a same source. As we illustrated in Section ?, such a case is disallowed for unambiguity.

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

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

Fig. 2. Subtyping.

3.3 Small-step Semantics and Congruence

Figure ? and Figure ? define small-step semantic rules 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 (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. The three rules require that the receiver and the arguments have been evaluated into values, before substitution is applied.

3.4 Auxilary Definitions

To make our formalization concise and expressive, we have defined a list of auxiliary functions, collected by Figure 6. To begin with, we introduce the basic functions: `ext`, `updateSet` and `prune`. $\texttt{ext}(I, J)$ simply indicates that interface I directly extends interface J. Corresponding to this is a more general case $I <: J$, meaning that I is a subtype of J. $\texttt{updateSet}(I, J)$ returns a set of methods defined in I that have "`override` J" in their signatures. Notice that $\texttt{updateSet}(I, I)$ 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.

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

$$(\textsc{T-Invk})\ \dfrac{\Gamma \vdash e_0 : \mathrm{I}_0 \qquad \texttt{mtype}(m, \mathrm{I}_0) = \overline{\mathrm{J}} \to \mathrm{I} \qquad \Gamma \vdash \overline{e} : \overline{\mathrm{I}} \qquad \overline{\mathrm{I}} <: \overline{\mathrm{J}}}{\Gamma \vdash e_0.m(\overline{e}) : \mathrm{I}}$$

$$(\textsc{T-PathInvk})\ \dfrac{\Gamma \vdash e_0 : \mathrm{I}_0 \qquad \mathrm{I}_0 <: \mathrm{J}_0 \qquad \texttt{mtype}(m, \mathrm{J}_0) = \overline{\mathrm{J}} \to \mathrm{I} \qquad \Gamma \vdash \overline{e} : \overline{\mathrm{I}} \qquad \overline{\mathrm{I}} <: \overline{\mathrm{J}}}{\Gamma \vdash e_0.\mathrm{J}_0 :: m(\overline{e}) : \mathrm{I}}$$

$$(\textsc{T-SuperInvk})\ \dfrac{\begin{array}{c}\Gamma \vdash \mathrm{this} : \mathrm{I}_0 \\ \texttt{ext}(\mathrm{I}_0, \mathrm{J}_0) \qquad \texttt{mtype}(m, \mathrm{J}_0) = \overline{\mathrm{J}} \to \mathrm{I} \qquad \Gamma \vdash \overline{e} : \overline{\mathrm{I}} \qquad \overline{\mathrm{I}} <: \overline{\mathrm{J}}\end{array}}{\Gamma \vdash \texttt{super}.\mathrm{J}_0 :: m(\overline{e}) : \mathrm{I}}$$

$$(\textsc{T-New})\ \Gamma \vdash \texttt{new}\ \mathrm{I}() : \mathrm{I}$$

$$(\textsc{T-Method})\ \dfrac{\begin{array}{c}\Gamma \vdash \mathrm{this} : \mathrm{I} \qquad \mathrm{I} <: \mathrm{J} \qquad \Gamma, \overline{x} : \overline{\mathrm{I}_X} \vdash e_0 : \_ <: \mathrm{I}_E \\ \texttt{mostSpecific}(m, \mathrm{I}, \mathrm{J}) = \{\mathrm{J}\} \qquad \texttt{mtype}(m, \mathrm{J}) = \overline{\mathrm{I}_X} \to \mathrm{I}_E\end{array}}{\mathrm{I}_E\ m(\overline{\mathrm{I}_X\ x})\ \texttt{override}\ \mathrm{J}\ \{\texttt{return}\ e_0;\}\ \mathrm{OK\ IN\ I}}$$

$$(\textsc{T-Intf})\ \dfrac{\begin{array}{c}\overline{\mathrm{I}}\ \mathrm{OK} \qquad \overline{\mathrm{M}}\ \mathrm{OK\ IN\ I} \\ \mathrm{TODO} : \textit{unambiguity check}(\textit{forall}\ m, \mathrm{J} < \mathrm{I}, \textit{mbody}(m, \mathrm{I}, \mathrm{J})\ \textit{defined?})\end{array}}{\texttt{interface}\ \mathrm{I}\ \texttt{extends}\ \overline{\mathrm{I}}\ \{\overline{\mathrm{M}}\}\ \mathrm{OK}}$$

Fig. 3. Typing rules.

$$(\textsc{S-Invk})\ \dfrac{\texttt{mbody}(m, \mathrm{I}, \mathrm{J}) = (\overline{\mathrm{I}_X}\ \overline{x}, \mathrm{I}'_E\ e_0)}{(<\mathrm{J}>\ \texttt{new}\ \mathrm{I}()).m(\overline{<\mathrm{I}_E>e}) \to <\mathrm{I}'_E> [\overline{<\mathrm{I}_X>e}/\overline{x}, <\mathrm{J}>\ \texttt{new}\ \mathrm{I}()/\texttt{this}]e_0}$$

$$(\textsc{S-PathInvk})\ \dfrac{\texttt{mbody}(m, \mathrm{I}, \mathrm{K}) = (\overline{\mathrm{I}_X}\ \overline{x}, \mathrm{I}'_E\ e_0)}{(<\mathrm{J}>\ \texttt{new}\ \mathrm{I}()).\mathrm{K} :: m(\overline{<\mathrm{I}_E>e}) \to <\mathrm{I}'_E> [\overline{<\mathrm{I}_X>e}/\overline{x}, <\mathrm{J}>\ \texttt{new}\ \mathrm{I}()/\texttt{this}]e_0}$$

$$(\textsc{S-SuperInvk})\ \dfrac{\texttt{mbody}(m, \mathrm{K}, \mathrm{K}) = (\overline{\mathrm{I}_X}\ \overline{x}, \mathrm{I}'_E\ e_0)}{\texttt{super}.\mathrm{K} :: m(\overline{<\mathrm{I}_E>e}) \to <\mathrm{I}'_E> [\overline{<\mathrm{I}_X>e}/\overline{x}, <\mathrm{J}>\ \texttt{new}\ \mathrm{I}()/\texttt{this}]e_0}$$

Fig. 4. Small-step semantics.Haoyuan: TODO: <???> new I()/this

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

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

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

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

$$(\textsc{C-StaticType})\ \texttt{new}\ I() \to < I > \texttt{new}\ I()$$

$$(\textsc{C-FReduce})\ \frac{e \to e' \qquad e \neq \texttt{new}\ I()}{< I > e \to < I > e'}$$

$$(\textsc{C-AnnoReduce})\ < I > (< J > \texttt{new}\ K()) \to < I > \texttt{new}\ K()$$

Fig. 5. Congruence.

$$\frac{\texttt{mostSpecific}(m, I_d, I_s) = \{I\} \qquad \texttt{mostSpecific}_2(m, I_d, I) = \{J\} \qquad \texttt{interface}\ J\ \texttt{extends}\ \overline{J}\ \{I_E\ m(\overline{I_X\ x})\ \texttt{override}\ I\ \{\texttt{return}\ e_0;\}\ldots\}}{\texttt{mbody}(m, I_d, I_s) = (\overline{I_X}\ \overline{x}, I_E\ e_0)}$$

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

$$\frac{set = \{K <: J \text{ and } K >: I \mid m \in \texttt{updateSet}(K, J)\}}{\texttt{mostSpecific}_2(m, I, J) = prune(set)}$$

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

$$\frac{\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{I_E\ m(\overline{I_X\ x})\ \texttt{override}\ J\ldots\}}{m \in \texttt{updateSet}(I, J)}$$

Fig. 6. Auxiliary functions.

`mostSpecific` and `mostSpecific`$_2$ `mostSpecific` is an auxiliary function that finds the most specific original implementations of a method. Let us consider `mostSpecific`(m, I, 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 ?, an interface belongs to the return set if and only if:

- 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, as the most specific one is always in the sub-most type.

We could have put $\mathit{set1}$ and $\mathit{set2}$ together, but the current definition leads a clearer illustration.

The `mostSpecific` function only focuses on original method implementations, all the method updates are omitted during that time. On the other hand, another auxiliary function `mostSpecific`$_2(m, I, J)$ has the assumption that J defines an original m, and this function tries to find the most specific implementations that update such an m. Just as `mostSpecific`, `mostSpecific`$_2$ 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 defines a method update for $J.m$;
- It does not have a subtype in the same set.

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

`mbody` and `mtype` H: mtype based on mbody? `mbody`(m, I_d, I_s), as defined in Figure 6, denotes a method body lookup function. We use I_d, I_s, since `mbody` is usually invoked by a receiver of a method m, with its dynamic type I_d and static type I_s. Such a function returns the most specific method implementation, more accurately, its parameters, returned expression and the types. It considers both originally defined methods and method updates, so `mostSpecific` and `mostSpecific`$_2$ are invoked.

To calculate `mbody`(m, I_d, I_s):

- First, it invokes `mostSpecific`(m, I_d, I_s) and returns a set.
- If `mostSpecific` returns a singleton set $\{I\}$, then it is good, otherwise `mbody` is undefined in this case. The set $\{I\}$ implies that we will use the m from I without ambiguity. But moreover, we have to invoke `mostSpecific`$_2(m, I_d, I)$, to check if there are updated versions of m between I_d and I. Again we forbid ambiguity, so the expected set after pruning is also a singleton set $\{J\}$.

– Finally, we fetch the implementation of $\mathtt{m}$ in interface J and return its related information.

The definition of `mtype` simply relies on `mbody`. In short,

$$\mathtt{mbody}(\mathtt{m}, \mathrm{I}, \mathrm{I}) = (\overline{\mathrm{I}_x}\ \overline{x}, \mathrm{I}_E\ e) \implies \mathtt{mtype}(\mathtt{m}, \mathrm{I}) = \overline{\mathrm{I}_x} \to \mathrm{I}_E$$

3.5 Type Safety

Lemma 1. *If* $\mathit{mtype}(\mathtt{m}, \mathrm{I}, \mathrm{J}) = \overline{\mathrm{D}} \to \mathrm{D}$, *and* $\mathit{mbody}(\mathtt{m}, \mathrm{I}, \mathrm{J}) = \overline{x}.e$, *then for some* J_0 *with* $\mathrm{I} <: \mathrm{J}_0$, $\exists \mathrm{C} <: \mathrm{D}$, *such that* $\overline{x} : \overline{\mathrm{D}}, \mathit{this} : \mathrm{J}_0 \vdash e : \mathrm{C}$.

Proof.
The base case: if $\mathtt{m}$ *is defined in* I, *then it is easy since* $\mathtt{m}$ *is defined in* I *and* $\overline{x} : \overline{\mathrm{D}}, \mathtt{this} : \mathrm{I} \vdash e : \mathrm{C}$ *by (T-METHOD). The induction step is also straightforward.*
□

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

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

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

Proof.
Case Var. $e = x \quad \mathrm{I} = \Gamma(x)$.
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 $\mathrm{I} = \mathrm{I}_{X_i}$, then since $[\overline{y}/\overline{x}]x = [\overline{y}/\overline{x}]x_i = y_i$, letting $\mathrm{I}' = \mathrm{I}_{Y_i}$ finishes the case.
Case New. $e = \mathtt{new}\ \mathrm{I}()$ and there are no term for substitution, the conclusion is obvious.
Case Invk. $e = e_0.\mathtt{m}(\overline{e}) \quad \Gamma, \overline{x} : \overline{\mathrm{I}_X} \vdash e_0 : \mathrm{I}_0$

$$\mathtt{mtype}(\mathtt{m}, \mathrm{I}_0) = \overline{\mathrm{I}_E} \to \mathrm{J}$$

$$\Gamma, \overline{x} : \overline{\mathrm{I}_X} \vdash \overline{e} : \overline{\mathrm{I}} \quad \overline{\mathrm{I}} <: \overline{\mathrm{I}_E}$$

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

$$\Gamma \vdash [\overline{y}/\overline{x}]e_0 : \mathrm{I}_0' \quad \mathrm{I}_0' <: \mathrm{I}_0$$

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

By lemma 2, $\mathtt{mtype}(\mathtt{m}, \mathrm{I}_0') = \overline{\mathrm{I}_E} \to \mathrm{J}$, then $\overline{\mathrm{I}_E'} <: \overline{\mathrm{I}_E}$ by the transitivity of $<:$. Therefore, by the rule (T-INVK), $\Gamma \vdash [\overline{y}/\overline{x}]e_0.\mathtt{m}([\overline{y}/\overline{x}]\overline{e}) : \mathrm{J}$.
Case PathInvk. $e = e_0.\mathrm{I} :: \mathtt{m}(\overline{e})$ and proof is similar as case Var.
Case SuperInvk. $e = \mathtt{super}.\mathrm{I} :: \mathtt{m}(\overline{e})$
Suppose $\Gamma \vdash \mathtt{this} : \mathrm{I}_0$, the following proof should be similar as case Var. □

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

Proof.
Case Invk. let

$$\mathsf{e} =< \mathsf{J} > \texttt{new}\ \mathrm{I}().\mathsf{m}(< \overline{\mathrm{I}_\mathsf{E}} > \overline{\mathsf{e}})$$

Suppose

$$\texttt{mbody}(\mathsf{m}, \mathrm{I}, \mathsf{J}) = (\overline{\mathrm{I}_\mathsf{X}}\ \overline{\mathsf{x}}, \mathrm{I}_{\mathsf{E}_0}\ \mathsf{e}_0)$$

then

$$\mathsf{e}' = [< \overline{\mathrm{I}_\mathsf{X}} > \overline{\mathsf{e}}/\overline{\mathsf{x}},\ < \mathsf{J} > \texttt{new}\ \mathrm{I}()/\texttt{this}]\mathsf{e}_0$$

By rules (T-NEW) and (T-INVK),

$$\Gamma \vdash \texttt{new}\ \mathrm{I}() : \mathrm{I} \quad \texttt{mtype}(\mathsf{m}, \mathrm{I}, \mathsf{J}) = \overline{\mathrm{I}_\mathsf{X}} \rightarrow \mathrm{I}_{\mathsf{E}_0} \quad \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{Y}_\mathsf{E}} \quad \overline{\mathsf{Y}_\mathsf{E}} <: \overline{\mathsf{X}} \quad \textit{, for some } \overline{\mathsf{Y}_\mathsf{E}}$$

By Lemma 1,

$$\Gamma, \overline{\mathsf{x}} : \overline{\mathrm{I}_\mathsf{X}}, \texttt{this} : \mathsf{J}_0 \vdash \mathsf{e}_0 : \mathrm{I}_\mathsf{F}, \textit{ for some } \mathsf{J} <: \mathsf{J}_0 \textit{ and } \mathrm{I}_\mathsf{F} <: \mathrm{I}_{\mathsf{E}_0}$$

By Lemma 3,

$$\Gamma \vdash [< \overline{\mathrm{I}_\mathsf{X}} > \overline{\mathsf{e}}/\overline{\mathsf{x}},\ < \mathsf{J} > \texttt{new}\ \mathrm{I}()/\texttt{this}]\mathsf{e}_0 : \mathrm{I}_\mathsf{G}, \textit{ for some } \mathrm{I}_\mathsf{G} <: \mathrm{I}_\mathsf{F}$$

So $\mathrm{I}_\mathsf{G} <: \mathrm{I}_{\mathsf{E}_0}$, finally just let $\mathrm{I}' = \mathrm{I}_\mathsf{G}$.
Case PathInvk. let

$$\mathsf{e} =< \mathsf{J} > \texttt{new}\ \mathrm{I}().\mathsf{K} :: \mathsf{m}(< \overline{\mathrm{I}_\mathsf{E}} > \overline{\mathsf{e}})$$

Suppose

$$\texttt{mbody}(\mathsf{m}, \mathrm{I}, \mathsf{K}) = (\overline{\mathrm{I}_\mathsf{X}}\ \overline{\mathsf{x}}, \mathrm{I}_{\mathsf{E}_0}\ \mathsf{e}_0)$$

then

$$\mathsf{e}' = [< \overline{\mathrm{I}_\mathsf{X}} > \overline{\mathsf{e}}/\overline{\mathsf{x}},\ < \mathsf{K} > \texttt{new}\ \mathrm{I}()/\texttt{this}]\mathsf{e}_0$$

By rules (T-NEW) and (T-INVK),

$$\Gamma \vdash \texttt{new}\ \mathrm{I}() : \mathrm{I} \quad \texttt{mtype}(\mathsf{m}, \mathrm{I}, \mathsf{K}) = \overline{\mathrm{I}_\mathsf{X}} \rightarrow \mathrm{I}_{\mathsf{E}_0} \quad \Gamma \vdash \overline{\mathsf{e}} : \overline{\mathsf{Y}_\mathsf{E}} \quad \overline{\mathsf{Y}_\mathsf{E}} <: \overline{\mathsf{X}} \quad \textit{, for some } \overline{\mathsf{Y}_\mathsf{E}}$$

By Lemma 1,

$$\Gamma, \overline{\mathsf{x}} : \overline{\mathrm{I}_\mathsf{X}}, \texttt{this} : \mathsf{J}_0 \vdash \mathsf{e}_0 : \mathrm{I}_\mathsf{F}, \textit{ for some } \mathsf{K} <: \mathsf{J}_0 \textit{ and } \mathrm{I}_\mathsf{F} <: \mathrm{I}_{\mathsf{E}_0}$$

By Lemma 3,

$$\Gamma \vdash [< \overline{\mathrm{I}_\mathsf{X}} > \overline{\mathsf{e}}/\overline{\mathsf{x}},\ < \mathsf{K} > \texttt{new}\ \mathrm{I}()/\texttt{this}]\mathsf{e}_0 : \mathrm{I}_\mathsf{G}, \textit{ for some } \mathrm{I}_\mathsf{G} <: \mathrm{I}_\mathsf{F}$$

So $\mathrm{I}_\mathsf{G} <: \mathrm{I}_{\mathsf{E}_0}$, finally just let $\mathrm{I}' = \mathrm{I}_\mathsf{G}$.
Case Super-Invk. let

$$\mathsf{e} = \texttt{super}.\mathsf{K} :: \mathsf{m}(< \overline{\mathrm{I}_\mathsf{E}} > \overline{\mathsf{e}})$$

Suppose

$$\texttt{mbody}(\mathtt{m}, \mathsf{K}, \mathsf{K}) = (\overline{\mathrm{I}_{\mathsf{X}}}\ \overline{\mathtt{x}}, \mathrm{I}_{\mathsf{E}_0}\ \mathtt{e}_0)$$

then

$$\mathtt{e}' = [< \overline{\mathrm{I}_{\mathsf{X}}} > \overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}_0$$

By rules (T-NEW) and (T-INVK),

$$\texttt{mtype}(\mathtt{m}, \mathsf{K}, \mathsf{K}) = \overline{\mathrm{I}_{\mathsf{X}}} \to \mathrm{I}_{\mathsf{E}_0} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathsf{Y}_{\mathsf{E}}} \quad \overline{\mathsf{Y}_{\mathsf{E}}} <: \overline{\mathsf{X}} \quad \textit{, for some } \overline{\mathsf{Y}_{\mathsf{E}}}$$

By Lemma 1,

$$\Gamma, \overline{\mathtt{x}} : \overline{\mathrm{I}_{\mathsf{X}}}, \texttt{this} : \mathsf{J}_0 \vdash \mathtt{e}_0 : \mathrm{I}_{\mathsf{F}}, \textit{ for some } \mathsf{K} <: \mathsf{J}_0 \textit{ and } \mathrm{I}_{\mathsf{F}} <: \mathrm{I}_{\mathsf{E}_0}$$

By Lemma 3,

$$\Gamma \vdash [< \overline{\mathrm{I}_{\mathsf{X}}} > \overline{\mathtt{e}}/\overline{\mathtt{x}}]\mathtt{e}_0 : \mathrm{I}_{\mathsf{G}}, \textit{ for some } \mathrm{I}_{\mathsf{G}} <: \mathrm{I}_{\mathsf{F}}$$

So $\mathrm{I}_{\mathsf{G}} <: \mathrm{I}_{\mathsf{E}_0}$, finally just let $\mathrm{I}' = \mathrm{I}_{\mathsf{G}}$.

□

Theorem 2 (Progress). *If* $\vdash \mathtt{e} : \mathrm{I}$, *then either* $\mathtt{e}$ *is a value or there is an* $\mathtt{e}'$ *with* $\mathtt{e} \to \mathtt{e}'$.

Theorem 3 (Type Soundness). $\mathtt{aa}$

4 Implementation

The prototype is implemented in Scala.

5 Discussion & Limitation

- Design space
- ??? (for example: ambiguity?)

6 Related Work

- Static+Dynamic type method lookup (any existing language that supports this?)
- Formalization based on FJ (novelty: keep static types <I> in formalization)
  - Existing formalizations based on FJ proposed new features and added rules in syntax and semantics. But we not only add rules, but also piggyback static types on almost all semantic rules to model method lookup.
  - Featherweight defenders, ...

6.1 Mainstream Multiple Inheritance Models

Multiple inheritance is a useful feature in object-oriented programming world although it's difficult to model and can cause various problems (e.g. the diamond problem). There are many existing languages/models that support multiple inheritance, either coming with multiple inheritance capability or added through evolution. The Mixin model 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.

Simplifying the mixins approach, traits [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. The introduction of default methods opens the gate for various flavors of multiple inheritance in Java.

Malayeri and Aldrich proposed a model CZ [5] 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 confliction in the same way: requiring programmers to explicitly resolve ambiguity and disallowing two methods with the same signature from two different libraries to co-exit.

6.2 Resolving Unintended Method Confliction

There are still a few languages that already realized the problem and explored a bit. We discuss them one by one.

C++ model. As discussed in Section **??**, C++ supports very flexible inheritance mode and allows programmers to choose different confliction resolution approaches via normal or virtual inheritance. Generally speaking, C++ model is much more complex and flexible than our approach. For example, given the following code

```
class A { public: void m() {cout << "MA" << endl;}};
class B { public: void m() {cout << "MB" << endl;}};
class C : public A, public B {
  void m() {cout << "MC" << endl;}
};
void func(A* a) { a->m(); }
int main() {
  C* c = new C();
  c->B::m();
  func(c);
  return 0; //Running result: MB MA
}
```

The running result is $\mathsf{MB}\ \mathsf{MA}$, meaning that it looks at the static type when doing method lookup. However, we can alter the code a little bit with virtual method and the result will be totally different:

```
class A {
  public: virtual void m() {cout << "MA" << endl;}
};
class B {
  public: virtual void m() {cout << "MB" << endl;}
};
class C : public A, public B {
    public: virtual void m() {cout << "MC" << endl;}
};
void func(A* a) { a->m(); }
int main() {
  C* c = new C();
  c->B::m();
  func(c);
  return 0; //Running result: MB MC
}
```

Now the running result will be $\mathsf{MB}\ \mathsf{MC}$. With virtual method, method lookup algorithm will find the most specific method definition of m, which is the definition in class C. Although C++ support this flexibility, it does not support updating both $\mathsf{A.m}$ and $\mathsf{B.m}$ in class C.

C# Explicit method implementation. Explicit method implementation is a special feature supported by C# . As described in C# tutorial [], 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: Explicit interface member implementations 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. Explicit interface member implementations allow disambiguation of interface members with the same signature. This is the the advantage that is similar to ours' model. However, there are two diffirencies: firstly, method (default) implementations are not allowed in C# interfaces; secondly, when handling unintended method confliction, multi-way method updating is not supported in C# .

Self. In the prototype-based language **SELF** [1], 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 than class-based languages, the multiple inheritance model is somehow similar. The **SELF** language support multiple (object) inheritance in a clever way. Not only the new inheritance relation *prioritized parents* is developed, but also the method lookup rule called *sender path tiebreaker rule*, which is similar with our approach, using hirarchical information in method resolution, but in a prototype-based language setting.

6.3 Static+Dynamic Type Method Lookup

6.4 Formalization Based on FeatherweightJava

FeatherweightJava [3] is a minimal core calculus of Java language, proposed by Igarashi et. al. Many models are based on FeatherweightJava, for example FeatherTrait [4], Featherweight defenders [2], Jx [6], as well as our model. It provides the standard model of formalizing a Java-like object-oriented language and people can easily extends it to form a new language. In terms of formalization, they 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 appear in literature before.

7 Conclusion

This paper proposes **MIM** 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 **MIM** 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 binding. This paper also introduces method updates for refinements on branches individually. **MIM** is formalized with a basis on Featherweight Java, and supported by a few theorems.

Our model can certainly be improved at some aspects. We did not formalize fields, though it is difficult to put state and multiple inheritance together []. For simplicity, we did not formalize other common features including method overloading, casts, covariant return types, and so on. Moreover, we restrict method updates to target only at the original methods. Potentially a looser condition can better support encapsulation and modularity with respect to code design. HAOYUAN: aaa BRUNO: bbb

References

1. 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)
2. 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)
3. 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)
4. Liquori, L., Spiwack, A.: Feathertrait: A modest extension of featherweight java. ACM Trans. Program. Lang. Syst. 30(2), 11 (2008)
5. Malayeri, D., Aldrich, J.: Cz: Multiple inheritance without diamonds. In: OOPSLA '09 (2009)
6. 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)

7. Scharli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. In: ECOOP'03 (2003)

A Appendix

A.1 A Non-Trivial Example

```
interface A extends {
A m() override A {return new A(); }
}
interface B extends A {
A m() override B {return new B(); }
}
interface C extends A {
A m() override C {return new C(); }
}
interface D extends B, C {
A m() override B {return new D(); }
}
interface E extends B {
A m() override B {return new E(); }
}
interface F extends D, E {
A m() override B {return new F(); }
A n(B b) override F {return b.m(); }
}
new F().n(new F())
```

Unfortunately I think this example shows that it is hard to reuse $\mathsf{D.m}$ on path B and $\mathsf{E.m}$ on path B in F?

We can use the `super` keyword to access the originally defined methods in super types, but we cannot access the old updating methods.

Just like $\texttt{super}.\mathsf{I} :: \mathsf{m}()$ is equivalent to $\texttt{new}\ \mathsf{I}().\mathsf{m}()$, maybe we can add a degree of freedom to `super`, for example, $\texttt{super}.\mathsf{D} :: \mathsf{B} :: \mathsf{m}()$ is equivalent to $\texttt{new}\ \mathsf{D}().\mathsf{B} :: \mathsf{m}()$, so we can use $\texttt{super}.\mathsf{D} :: \mathsf{B} :: \mathsf{m}()$ and $\texttt{super}.\mathsf{E} :: \mathsf{B} :: \mathsf{m}()$ inside interface F for code reuse?

Interfaces $\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}$ OK in type checking.

To type-check $\texttt{new}\ \mathsf{F}().\mathsf{n}(\texttt{new}\ \mathsf{F}())$:

- By (T-INVK), we need to calculate $\texttt{mtype}(\mathsf{n}, \mathsf{F})$.
- $\texttt{mtype}(\mathsf{n}, \mathsf{F}) = \mathsf{B} \to \mathsf{A}$. And $\texttt{new}\ \mathsf{F}() : \mathsf{B}$.
- $\texttt{new}\ \mathsf{F}().\mathsf{n}(\texttt{new}\ \mathsf{F}()) : \mathsf{A}$.

To compute $\texttt{new}\ \mathsf{F}().\mathsf{n}(\texttt{new}\ \mathsf{F}())$:

- By (C-RECEIVER), we compute $\texttt{new}\ \mathsf{F}()$:
  - By (C-STATICTYPE): $< \mathsf{F} > \texttt{new}\ \mathsf{F}()$.
- By (C-ARGS), we compute $\texttt{new}\ \mathsf{F}()$:
  - By (C-STATICTYPE): $< \mathsf{F} > \texttt{new}\ \mathsf{F}()$.
- Now we get $(< \mathsf{F} > \texttt{new}\ \mathsf{F}()).\mathsf{n}(< \mathsf{F} > \texttt{new}\ \mathsf{F}())$. By (S-INVK):
  - Compute $\texttt{mbody}(\mathsf{n}, \mathsf{F}, \mathsf{F}) = (\mathsf{B}\ \mathsf{b}, \mathsf{A}\ \mathsf{b}.\mathsf{m}())$.
  - Replace b with $< \mathsf{B} > \texttt{new}\ \mathsf{F}()$ in $\mathsf{b}.\mathsf{m}()$.
  - Replace `this` with $< \mathsf{F} > \texttt{new}\ \mathsf{F}()$ in $\mathsf{b}.\mathsf{m}()$.
- Finally we get $< \mathsf{A} > ((< \mathsf{B} > \texttt{new}\ \mathsf{F}()).\mathsf{m}())$.

- By (C-FREDUCE), we first compute $(<\mathsf{B}>\ \texttt{new}\ \mathsf{F}()).\mathsf{m}()$:
  - By (S-INVK), we compute $\texttt{mbody}(\mathsf{m}, \mathsf{F}, \mathsf{B})$.
  - In `mbody`, we invoke $\texttt{mostSpecific}(\mathsf{m}, \mathsf{F}, \mathsf{B})$.
  - In `mostSpecific`, $\mathsf{set} = \{\mathsf{B}\}$, $\texttt{prune}(\mathsf{set}) = \{\mathsf{B}\}$.
  - Back to `mbody`, we invoke $\texttt{mostSpecific}_3(\mathsf{m}, \mathsf{F}, \mathsf{B})$.
  - In $\texttt{mostSpecific}_3$, $\mathsf{set} = \{\mathsf{B}, \mathsf{D}, \mathsf{E}, \mathsf{F}\}$, $\texttt{prune}(\mathsf{set}) = \{\mathsf{F}\}$.
  - Back to `mbody`, we check $\mathsf{F.m}$ and return $(-, \mathsf{A}\ (\texttt{new}\ \mathsf{F}()))$.
  - Back to (S-INVK).
  - Replace $\texttt{this}$ with $<\mathsf{B}>\ \texttt{new}\ \mathsf{F}()$ in $\texttt{new}\ \mathsf{F}()$.
  - Finally we get $<\mathsf{A}>\ \texttt{new}\ \mathsf{F}()$.
- Now we have $<\mathsf{A}>(<\mathsf{A}>\ \texttt{new}\ \mathsf{F}())$.
- By (C-ANNOREDUCE): $<\mathsf{A}>\ \texttt{new}\ \mathsf{F}()$.

TODO: class encapsulation problem: interface A: m; interface B: m update A; interface C: m update B.