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 Overview

- Code for the problem (Draw example)
- Potential fixes (renaming, traits, ...)
- How it is solved with our language (client code, simple explanation) + show update

```
interface Deck {
   draw();  // draws a card from the Deck
}

interface Drawable {
   draw();  // draws something on the screen
}

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

2.1 Potential fixes

- Explicit override m() in C.
- Choose one (such as Mixin model).
- With Traits: method exclusion.
- With Traits: method rename.

In previous solutions, only renaming can preserve the two different methods m(), but it's cumbersome to do in practice: new names have to be introduced, and (probably) need to prefer one over another. Other solutions only preserve one method. Can we keep both methods without renaming?

2.2 Solution in our language

```
interface Deck {
   void draw() { // draws a card from the Deck }
}
interface Drawable {
   void draw() { // draws something on the screen }
}
interface DrawableDeck extends Deck, Drawable {
  //both draw() methods are kept automaticlaly in our language
}
new DrawableDeck().Deck::draw()
```

2.3 Method refinement

```
interface DrawableDeck2 extends DrawableDeck {
   void draw() override Deck {
      // new implementation of draw a card from deck
   }
   void draw() override Drawable {
      // new implementation of drawing something
```

```
    }
}
new DrawableDeck2().Deck::draw()
```

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` 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 "$e.\mathsf{I} :: m(\overline{e})$", 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 "$< \mathsf{I} >$ `new` 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.

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,

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

Interface names I, J, K
Method names m
Variable names x

Fig. 1. Syntax. *: only intended for semantic rules.

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.

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

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

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

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

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

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

$$\text{(T-SuperInvk)}\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{this} : I_0 \\ \mathtt{ext}(I_0, J_0) \qquad \mathtt{mtype}(m, J_0) = \overline{J} \to I \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J}\end{array}}{\Gamma \vdash \mathtt{super}.J_0 :: m(\overline{e}) : I}$$

$$\text{(T-New)}\ \Gamma \vdash \mathtt{new}\ I() : I$$

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

$$\text{(T-Intf)}\ \frac{\overline{I}\ \text{OK} \qquad \overline{M}\ \text{OK IN}\ I}{\mathtt{interface}\ I\ \mathtt{extends}\ \overline{I}\ \{\overline{M}\}\ \text{OK}}$$

Fig. 3. Typing rules.

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 $\texttt{mbody}(\mathfrak{m}, \mathsf{I}, \mathsf{J})$ in a way that it finds the most specific body of method $\mathfrak{m}$, 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.

$$\text{(S-Invk)}\ \frac{\texttt{mbody}(\mathfrak{m}, \mathsf{I}, \mathsf{J}) = (\overline{\mathsf{I}_X}\ \overline{x}, \mathsf{I}'_E\ e_0)}{(<\mathsf{J}>\texttt{new}\ \mathsf{I}()).\mathfrak{m}(\overline{<\mathsf{I}_E> e}) \rightarrow <\mathsf{I}'_E> [\overline{<\mathsf{I}_X> e}/\overline{x}, <\mathsf{J}>\texttt{new}\ \mathsf{I}()/\texttt{this}]e_0}$$

$$\text{(S-PathInvk)}\ \frac{\texttt{mbody}(\mathfrak{m}, \mathsf{I}, \mathsf{K}) = (\overline{\mathsf{I}_X}\ \overline{x}, \mathsf{I}'_E\ e_0)}{(<\mathsf{J}>\texttt{new}\ \mathsf{I}()).\mathsf{K} :: \mathfrak{m}(\overline{<\mathsf{I}_E> e}) \rightarrow <\mathsf{I}'_E> [\overline{<\mathsf{I}_X> e}/\overline{x}, <\mathsf{J}>\texttt{new}\ \mathsf{I}()/\texttt{this}]e_0}$$

$$\text{(S-SuperInvk)}\ \frac{\texttt{mbody}(\mathfrak{m}, \mathsf{K}, \mathsf{K}) = (\overline{\mathsf{I}_X}\ \overline{x}, \mathsf{I}'_E\ e_0)}{\texttt{super}.\mathsf{K} :: \mathfrak{m}(\overline{<\mathsf{I}_E> e}) \rightarrow <\mathsf{I}'_E> [\overline{<\mathsf{I}_X> e}/\overline{x}, <\mathsf{J}>\texttt{new}\ \mathsf{I}()/\texttt{this}]e_0}$$

Fig. 4. Small-step semantics.

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`. $\texttt{ext}(\mathsf{I}, \mathsf{J})$ simply indicates that interface I directly extends interface J. Corresponding to this is a more general case $\mathsf{I} <: \mathsf{J}$, meaning that I is a subtype of J. $\texttt{updateSet}(\mathsf{I}, \mathsf{J})$ returns a set of methods defined in I that have "`override` J" in their signatures. Notice that $\texttt{updateSet}(\mathsf{I}, \mathsf{I})$ 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.

`mostSpecific` and $\texttt{mostSpecific}_2$ `mostSpecific` is an auxiliary function that finds the most specific original implementations of a method. Let us consider $\texttt{mostSpecific}(\mathfrak{m}, \mathsf{I}, \mathsf{J})$, what it returns is a set of interfaces, each including its own $\mathfrak{m}$ 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 $\mathfrak{m}$;
- It is a supertype of I;

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

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

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

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

$$\text{(C-StaticType)}\ \mathsf{new}\ I() \to < I > \mathsf{new}\ I()$$

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

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

Fig. 5. Congruence.

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

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

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

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

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

Fig. 6. Auxiliary functions.

– 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 *set1* and *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 $\texttt{mostSpecific}_2(\mathfrak{m}, \mathrm{I}, \mathrm{J})$ has the assumption that J defines an original $\mathfrak{m}$, and this function tries to find the most specific implementations that update such an $\mathfrak{m}$. Just as `mostSpecific`, $\texttt{mostSpecific}_2$ 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 $\mathrm{J}.\mathfrak{m}$;
– 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` $\texttt{mbody}(\mathfrak{m}, \mathrm{I_d}, \mathrm{I_s})$, as defined in Figure 6, 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 $\mathfrak{m}$, 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 method updates, so `mostSpecific` and $\texttt{mostSpecific}_2$ are invoked.

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

– First, it invokes $\texttt{mostSpecific}(\mathfrak{m}, \mathrm{I_d}, \mathrm{I_s})$ 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 $\mathfrak{m}$ from I without ambiguity. But moreover, we have to invoke $\texttt{mostSpecific}_2(\mathfrak{m}, \mathrm{I_d}, \mathrm{I})$, to check if there are updated versions of $\mathfrak{m}$ between $\mathrm{I_d}$ and I. Again we forbid ambiguity, so the expected set after pruning is also a singleton set {J}.
– Finally, we fetch the implementation of $\mathfrak{m}$ in interface J and return its related information.

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

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

– Syntax + typing rules + Semantics + Auxiliary definitions
– Implementation: a simple type checker + interpreter in Scala

```
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 D.m on path B and 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 super.I :: m() is equivalent to new I().m(), maybe we can add a degree of freedom to super, for example, super.D :: B :: m() is equivalent to new D().B :: m(), so we can use super.D :: B :: m() and super.E :: B :: m() inside interface F for code reuse?

Interfaces A, B, C, D, E, F OK in type checking.

To type-check new F().n(new F()):

- By (T-INVK), we need to calculate mtype(n, F).
- mtype(n, F) = B → A. And new F() : B.
- new F().n(new F()) : A.

To compute new F().n(new F()):

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

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

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

3.5 Type Safety

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* $\mathrm{I}' <: \mathrm{I}$.

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

4 Implementation

The prototype is implemented in Scala.

5 Discussion & Limitation

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

6 Proof

7 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, ...

7.1 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.

7.2 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 MB 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 MB 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 A.m and 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.

7.3 Static+Dynamic Type Method Lookup

7.4 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.

8 Conclusion

Conclusion.

Acknowledgments

Authors would like to thank YYYYY.

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)