Introducing virtual field and rename dispatching based on view stack to fix Eiffel's rename loophole

ANONYMOUS AUTHOR(S)

We discovered a loophole in Eiffel's field renaming mechanism when applied to the diamond problem of multiple inheritance. To fix the loophole we propose to abandon the renaming's reference identity semantics; we introduce a concept called virtual field, and propose two methods: the first method is manual fix with help from enhanced programming rules, e.g. direct virtual field access is only allowed in accessor methods (among other rules); and the second method is automatic, we introduce rename dispatching based on the object's runtime view stack, hence providing an improved solution to multiple inheritance (esp. for unplanned MI). All the source code of this paper can be found in the supplementary material: eiffel_rename.tgz.

Additional Key Words and Phrases: virtual field, rename dispatching, view stack, (unplanned) multiple inheritance (MI), diamond problem, Eiffel language, name clash resolution

1 MOTIVATION: THE DIAMOND PROBLEM

The most well known problem in multiple inheritance (MI) is the diamond problem, e.g. on wikipedia [1] it is described as:

> The "diamond problem" is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?

Actually in the real world engineering practice, for any *method*'s ambiguity e.g. foo(), it is relatively easy to resolve *by the programmers*:

- just *override* it in D.foo(), or
- explicitly use fully quantified method names, e.g. A.foo(), B.foo(), or C.foo().

The more difficult problem is how to handle the *data members (i.e. fields)* inherited from A: shall D have one joined copy or two separate copies of A's fields (or mixed fields with some are joined, and others separated)? For example, in C++ [Stroustrup 1991], the former is called virtual inheritance, and the latter is default (regular) inheritance. But C++ does not completely solve this problem, for example let's build an object model for PERSON, STUDENT, FACULTY, and RESEARCH_ASSISTANT in a university:

[1] https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

2023. 2475-1421/2023/1-ART1 $15.00
https://doi.org/

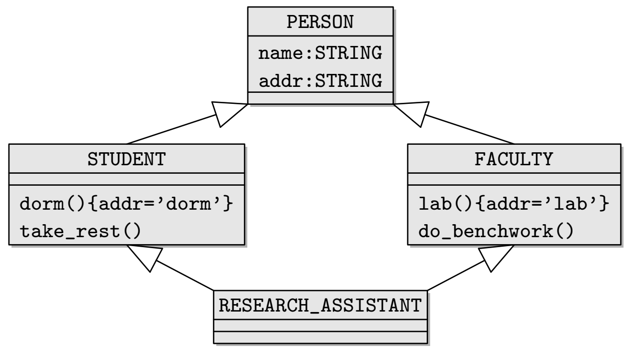

Fig. 1. the diamond problem in multiple inheritance

The intended semantics is that a RESEARCH_ASSISTANT should have only 1 name field, but 2 address fields: one "dorm" as STUDENT to take_rest(), and one "lab" as FACULTY to do_benchwork(); so in total 3 fields. However, in C++'s plain MI we can either do:

(1) virtual inheritance: RESEARCH_ASSISTANT will have 1 name, and 1 addr; in total 2 fields, or
(2) default inheritance: RESEARCH_ASSISTANT will have 2 names, and 2 addrs; in total 4 fields

Hence with C++'s plain MI mechanism, RESEARCH_ASSISTANT will have either one whole copy, or two whole copies of PERSON's all data members. This leaves something better to be desired.

Among all the OOP languages that support MI, Eiffel is unique in that it provides a renaming mechanism designed to resolve name clashes of each class member *individually* in the derived classes. Let's check how Eiffel models this example in the next Section 2, then we will examine the semantics of Eiffel's renaming mechanism in Section 2.2, and show the loophole we discovered in 2.3. In Section 3 we will fix the loophole with a manual approach, by introducing the concept of virtual fields and proposing some enhanced programming rules. In Section 4 we will introduce rename dispatching based on view stack to fix the loophole automatically with an experimental compiler, and describe our implementation in Section 5. In Section 6 we will discuss Eiffel's renamed methods, and compare virtual method dispatch and rename dispatching. In Section 7 we will discuss some related works. Finally, we summarize our work in Section 8.

2 EIFFEL'S RENAMING MECHANISM

(Note: all the code listings in this section are compilable and executable, so it's a bit verbose.)

To simulate *unplanned* [2] MI, let's assume each class is developed by different software vendors independently – hence uncoordinated, but in the topological order of inheritance directed acyclic graph (*DAG* for MI; while for single inheritance, it's inheritance *tree*), starting from the top base classes. And if class B inherits from A, we say B('s level) is *below* A.

The first vendor developed class PERSON, each person has a name and an address:

[2] By unplanned MI, we mean when a programmer works on class C, s/he cannot make any changes to any of Cs superclasses. So the programming language has to provide good feature adaptation mechanisms to allow the programmer of class C to achieve the intended semantics.

Listing 1. person.e (Eiffel)

```
class PERSON
inherit ANY redefine default_create end  -- needed by ISE and GOBO compiler; but not by SmartEiffel

feature {ANY}
   name: STRING
   addr: STRING

   get_addr():STRING  is do Result := addr end  -- accessor method, to read
   set_addr(a:STRING) is do addr   := a    end  -- accessor method, to write

   default_create is  -- the constructor
     do
       name := "name"
       addr := "addr"
     end
end
```

The second vendor developed class STUDENT: added set/get_student_addr() accessors, and take_rest() method, since PERSON has the `addr` field already, the second vendor just uses it instead of adding another field:

Listing 2. student.e (Eiffel)

```
class STUDENT
inherit PERSON

feature {ANY}
   get_student_addr():STRING  is do Result := get_addr() end  -- assign dorm semantics to addr
   set_student_addr(a:STRING) is do set_addr(a) end

   take_rest() is
     do
       io.put_string(name + " take_rest in the: " + get_student_addr() + "%N");
     end
end
```

At the *same time* as the second vendor, the third vendor developed class FACULTY: added set/get_faculty_addr() accessors, and do_benchwork() method, in the same way to reuse the inherited field `PERSON.addr` instead of adding another field:

Listing 3. faculty.e

```
class FACULTY
inherit PERSON

feature {ANY}
   get_faculty_addr():STRING  is do Result := get_addr() end  -- assign lab semantics to addr
   set_faculty_addr(a:STRING) is do set_addr(a) end

   do_benchwork() is
     do
       io.put_string(name + " do_benchwork in the: " + get_faculty_addr() + "%N");
     end
end
```

Definition 1 (semantic branching site of field)**.** At this point, we can see the two different inheritance branches of PERSON have assigned different semantics to the same inherited field `addr`, we call class PERSON as the *semantic branching site of field* `addr`.

By contrast, in the whole inheritance DAG of this example, there is no semantic branching site of field `name`.

2.1 Eiffel MI: individual feature renaming

With Eiffel language's renaming mechanism to treat each individual feature (class field or method) separately from the base classes, we implement RESEARCH_ASSISTANT as in listing 4.

Listing 4. research_assistant.e

```
class RESEARCH_ASSISTANT
inherit
        STUDENT rename addr as student_addr end  -- field student_addr inherit the dorm semantics
        FACULTY rename addr as faculty_addr end  -- field faculty_addr inherit the lab  semantics
        -- then select, NOTE: not needed by SmartEiffel, but needed by GOBO and ISE compiler
        PERSON  select addr end

create {ANY}
   make

feature {ANY}
   print_ra() is  -- print out all 3 addresses
     do
       io.put_string(name +" has 3 addresses: <"+ addr +", "+ student_addr +", "+ faculty_addr + ">%N")
     end

   make is  -- the constructor
     do
       name := "ResAssis"
       addr := "home"           -- the home semantics
       student_addr := "dorm"   -- the dorm semantics
       faculty_addr := "lab"    -- the lab  semantics
     end
end
```

Definition 2 (renaming site of a field)**.** If a class A has renamed any of its base class' field, we call A the *renaming site* of the field.

For example, RESEARCH_ASSISTANT is the renaming site for both STUDENT.addr and FACULTY.addr.

Actually we have made RESEARCH_ASSISTANT inherited from PERSON 3 times: 2 times indirectly via STUDENT and FACULTY, and 1 time directly from PERSON. Thus, RESEARCH_ASSISTANT has 1 `name` field (which is joined by default in Eiffel), and 3 address fields. The extra inheritance from PERSON is to make the inheritance from STUDENT and FACULTY symmetric, which helps easy exposition of the next Section 4 when we discuss view stacks.

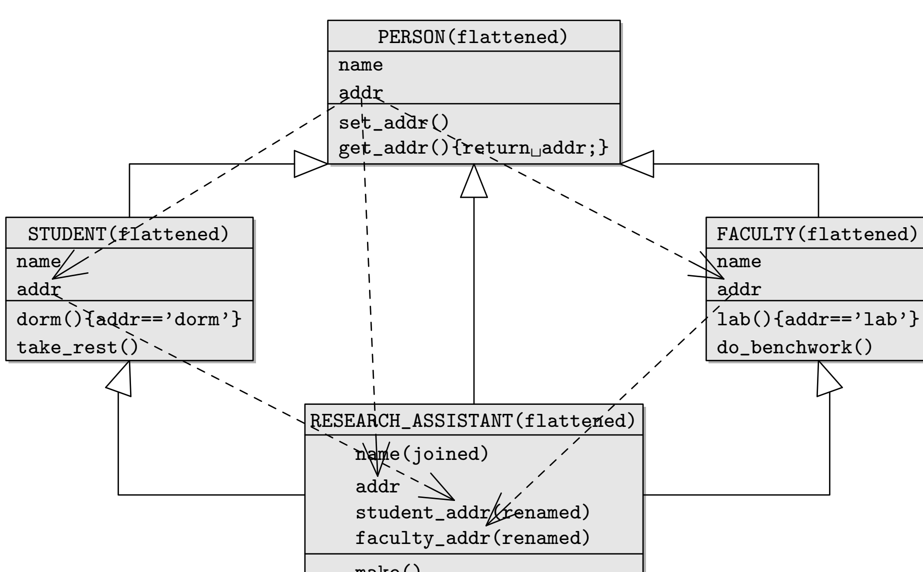

Fig. 2. flattened fields view, and feature renaming DAG of field 'addr' (dashed arrows)

The above diagram shows the field flattened view of the classes implemented in Eiffel; in particular, the dashed arrows show how the field `addr` is inherited (and renamed) in the class hierarchy, we call it *feature renaming DAG* of `addr`.

The purpose of the renaming mechanism here is to have separate copies for the name-clashed fields individually. Without renaming, all fields with the same name from the base classes are joined (i.e only one field with that name in the derived class, this is the default behavior of Eiffel, and we can see its application on the field `RESEARCH_ASSISTANT.name`). However for fields that need to be separated, joining will cause semantic error: e.g. `STUDENT.addr` v.s. `FACULTY.addr` in RESEARCH_ASSISTANT. For the inherited methods from STUDENT, they expect `addr` to have "dorm" semantics; while for the inherited methods from FACULTY, they expect `addr` to have "lab" semantics. But if these two fields with different semantics are joined in RESEARCH_ASSISTANT, they will share the same data member `addr`, it will cause disastrous bugs in the resulting program.

Note: strictly speaking the `select addr end` clause [3] on line 6 is not needed, because after the two renamings on line 3 and 4, there is no more name clash on RESEARCH_ASSISTANT's field `addr`, hence no ambiguity. However some Eiffel compilers (e.g. GOBO and ISE) enforce the presence of this `select` clause which we think is wrong, while others (e.g. SmartEiffel) do not.

2.2 Exam the semantics of the renamed fields

We want to study how the renamed field behaves in this diamond inheritance. In ECMA-367 [ECMA 2006] which serves as Eiffel language standard specification, Section 8.6.16, we only find a very brief description of its semantics:

> Renaming principle: Renaming does not affect the semantics of an inherited feature.

so we have to look further elsewhere: from [Meyer 1997] 15.2 page 544, we found some examples:

> ... we could have renamed both for symmetry:

```
class SANTA_BARBARA inherit
  LONDON
    rename foo as fog end
  NEW_YORK
    rename foo as zoo end
feature
  ...
end
```

on page 545:

> `l: LONDON; n: NEW_YORK; s: SANTA_BARBARA`
> Then l.foo and s.fog are both valid; after a polymorphic assignment l := s they would have the same effect, since the feature names represent the same feature. Similarly n.foo and s.zoo are both valid, and after n := s they would have the same effect.
> None of the following, however, is valid:
> - l.zoo, l.fog, n.zoo, n.fog since neither LONDON nor NEW_YORK has a feature called fog or zoo.

[3]The Eiffel `select` clause allows the programmer to *explicitly* resolve name clash on each inherited feature *individually*, which we believe is a better solution than imposing the *same* method resolution order (MRO) [Barrett et al. 1996] to *all* features as in many other OOP languages, e.g. Python [van Rossum 2010]: the base classes' order in the inheritance clause should *not* matter.

- s.foo since as a result of the renaming SANTA_BARBARA has no feature called foo.[4]

And on page 546, it's summarized as:

> Renaming is a syntactic mechanism, allowing you to refer to the same feature under different names in different classes.

from these descriptions esp. the last summary, we can conclude in Eiffel renaming is a *reference identity* relation between the original field and the new renamed field, let's use `<=>` to denote this relationship, i.e

- `LONDON.foo <=> SANTA_BARBARA.fog`, and
- `NEW_YORK.foo <=> SANTA_BARBARA.zoo`

So for our MI diamond problem example, the relationships are:

- `PERSON.addr = STUDENT.addr <=> RESEARCH_ASSISTANT.student_addr`
- `PERSON.addr = FACULTY.addr <=> RESEARCH_ASSISTANT.faculty_addr`

then it follows:

`RESEARCH_ASSISTANT.student_addr <=> RESEARCH_ASSISTANT.faculty_addr`

which means there is *no feature separation achieved* at all! We think this simple reference identity semantics of field renaming is a loophole in Eiffel, as demonstrated in this diamond problem. To further confirm our suspicion, we decided to verify it with the actual Eiffel compilers.

2.3 Verify the loophole with real Eiffel compilers

Let's create a RESEARCH_ASSISTANT object, and call the method do_benchwork() and take_rest() on it, and check the outputs:

Listing 5. app.e

```
-- to build with SmartEiffel: compile  app.e -o app
class APP inherit INTERNAL

create {ANY}
   make

feature {ANY}
   ra: RESEARCH_ASSISTANT
   p:  PERSON
   s:  STUDENT
   f:  FACULTY

   -- problematic implementation: direct field access
   print_student_addr_direct_field(u: STUDENT) is
      do io.put_string(u.name + " as STUDENT.addr: " + u.addr + "%N") end
   print_faculty_addr_direct_field(u: FACULTY) is
      do io.put_string(u.name + " as FACULTY.addr: " + u.addr + "%N") end

   -- correct implementation: use semantic assigning accessor
   print_student_addr_via_accessor(u: STUDENT) is
      do io.put_string(u.name + " as STUDENT.addr: " + u.get_student_addr() + "%N") end
   print_faculty_addr_via_accessor(u: FACULTY) is
      do io.put_string(u.name + " as FACULTY.addr: " + u.get_faculty_addr() + "%N") end

   make is
      do
         create  p.default_create
         create  s.default_create
         create  f.default_create
         create ra.make

         ra.print_ra()
         io.put_string("PERSON  size: " +physical_size(p ).out+ "%N")
```

[4]This example actually demonstrates why the **`select addr end`** clause in class RESEARCH_ASSISTANT is not necessary: after two renamings, there is no more name clash on `addr`.

```
        io.put_string("STUDENT size: " +physical_size(s ).out+ "%N")
        io.put_string("FACULTY size: " +physical_size(f ).out+ "%N")
        io.put_string("RESEARCH_ASSISTANT size: " +physical_size(ra).out+ "%N")

        ra.do_benchwork()  -- which addr field will this call access?
        ra.take_rest()     -- which addr field will this call access?

        io.put_string("-- print_student|faculty_addr_direct_field%N")
        print_student_addr_direct_field(ra)
        print_faculty_addr_direct_field(ra)

        io.put_string("-- print_student|faculty_addr_via_accessor%N")
        print_student_addr_via_accessor(ra)
        print_faculty_addr_via_accessor(ra)

        io.put_string("-- check reference identity%N")
        if                  ra.addr  = ra.faculty_addr
        then io.put_string("ra.addr  = ra.faculty_addr%N")
        else io.put_string("ra.addr != ra.faculty_addr%N") end

        if                  ra.addr  = ra.student_addr
        then io.put_string("ra.addr  = ra.student_addr%N")
        else io.put_string("ra.addr != ra.student_addr%N") end

        if                  ra.student_addr  = ra.faculty_addr
        then io.put_string("ra.student_addr  = ra.faculty_addr%N")
        else io.put_string("ra.student_addr != ra.faculty_addr%N") end

        io.put_string("-- test some assignment: suppose ra moved both lab2 and dorm2%N")
        ra.set_faculty_addr("lab2")
        ra.print_ra()
        ra.set_student_addr("dorm2")
        ra.print_ra()
      end
end
```

We have tested all the three major Eiffel compilers that we can find on the internet:

(1) the latest ISE EiffelStudio[5] 22.12.10.6463 (released in 2022)
(2) the open source Gobo Eiffel compiler gec version 22.01.09.4[6] (released in 2022)
(3) the open source GNU SmartEiffel version 1.1[7] (released in 2003).

All three compilers generate problematic outputs:

Listing 6. ISE EiffelStudio output: most of the lines are wrong

```
ResAssis has 3 addresses: <home, home, home>  -- all 3 addr is "home"! no feature separation at all!
PERSON  size: 32
STUDENT size: 32
FACULTY size: 32
RESEARCH_ASSISTANT size: 48
ResAssis do_benchwork in the: home
ResAssis take_rest in the: home
-- print_student|faculty_addr_direct_field
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- print_student|faculty_addr_via_accessor
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- check reference identity
ra.addr  = ra.faculty_addr
ra.addr  = ra.student_addr
ra.student_addr  = ra.faculty_addr
-- test assignment: suppose ra moved both lab2 and dorm2
ResAssis has 3 addresses: <lab2, lab2, lab2>
ResAssis has 3 addresses: <dorm2, dorm2, dorm2>
```

[5]https://www.eiffel.com/company/leadership/ the company with Eiffel language designer Dr. Bertrand Meyer as CEO and Chief Architect, Founder
[6]https://github.com/gobo-eiffel/gobo
[7]in later years, SmartEiffel was divergent from ECMA, so we choose to test version 1.1 (the last version of 1.x); and by convention: "Software Version 1.0 is used as a major milestone, indicating that the software has at least all major features plus functions the developers wanted to get into that version, and is considered reliable enough for general release." https://en.wikipedia.org/wiki/Software_versioning#Version_1.0_as_a_milestone

From ISE output, we can see our suspicion is confirmed. The output demonstrates a few problems:

(1) line 2 – line 5 show the object size in bytes, we can see indeed `RESEARCH_ASSISTANT` is bigger than both `STUDENT` and `FACULTY`, which means it has more data fields; while line 15 – 17 (check reference identity) [8], it demonstrates the renaming language construct does not help to achieve feature separation at all, this is a language loophole.

(2) Moreover, the 1st line of the ISE compiler output is three "home" strings! if we check the constructor `RESEARCH_ASSISTANT.make()` in listing 4, we can see the last assignment statement is `faculty_addr := "lab"`, even with reference identity semantics this output is wrong, is this an ISE compiler bug (also compare it with the line 19 – 20)?

(3) line 6 & 7, do_benchwork() and take_rest() all output "home", it failed to fulfill the programmer's intention.

(4) again, the last two lines 19 – 20, although it achieved the renaming's reference identity semantics, it does not achieve feature separation.

Listing 7. GOBO output

```
1  ResAssis has 3 addresses: <home, dorm, lab>
2  PERSON  size: 24
3  STUDENT size: 24
4  FACULTY size: 24
5  RESEARCH_ASSISTANT size: 40
6  ResAssis do_benchwork in the: home
7  ResAssis take_rest in the: home
8  -- print_student|faculty_addr_direct_field
9  ResAssis as STUDENT.addr: home  -- wrong addr read from ra object!
10 ResAssis as FACULTY.addr: home  -- wrong addr read from ra object!
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: home
13 ResAssis as FACULTY.addr: home
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test assignment: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <lab2, dorm, lab>   -- wrong addr set to ra object!
20 ResAssis has 3 addresses: <dorm2, dorm, lab>  -- wrong addr set to ra object!
```

From line 1 & 14 – 17, we can see GOBO compiler indeed separate the 3 address fields (is it a standard compliant compiler? i.e this shows it does *not* implement the reference identity semantics), but it failed to achieve the programmer's intention:

(1) line 6 & 7, do_benchwork() and take_rest() all output "home", same problem as ISE

(2) line 19 – 20, for assignment:
- `ra.set_faculty_addr("lab2")`, and
- `ra.set_student_addr("dorm2")`

instead it changed the value of `RESEARCH_ASSISTANT.addr`, while the very reason we introduced renaming is want to
- modify `RESEARCH_ASSISTANT.student_addr` on the `ra` object, and
- modify `RESEARCH_ASSISTANT.faculty_addr` on the `ra` object

Listing 8. SmartEiffel output

```
ResAssis has 3 addresses: <home, dorm, lab>
PERSON  size: 12
STUDENT size: 12
FACULTY size: 12
```

[8] In Eiffel, "=" means reference identity testing

```
RESEARCH_ASSISTANT size: 20
ResAssis do_benchwork in the: home
ResAssis take_rest in the: home
-- print_student|faculty_addr_direct_field
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- print_student|faculty_addr_via_accessor
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- check reference identity
ra.addr != ra.faculty_addr
ra.addr != ra.student_addr
ra.student_addr != ra.faculty_addr
-- test some assignment: suppose ra moved both lab2 and dorm2
ResAssis has 3 addresses: <lab2, dorm, lab>
ResAssis has 3 addresses: <dorm2, dorm, lab>
```

SmartEiffel's output is mostly the same as GOBO output (except the object size which is compiler dependent), and both compilers do *not* implement the reference identity semantics.

In particular, we can see for all the three compilers: the do_benchwork() method calls all print out the *problematic* address "home", while the programmer's intention is "lab"; and take_rest() print out "home" instead of "dorm".

Please also note: currently FACULTY.do_benchwork() calls FACULTY.get_faculty_addr(), and then PERSON.get_addr() to access the field **addr**; even if we change FACULTY.do_benchwork() or FACULTY.get_faculty_addr() to access the field **addr** directly, the output is still the same, which we have tested; and interested readers are welcome to verify it.

We choose to define these three pairs of seemingly redundant accessor methods:

- `PERSON.set/get_addr()`
- `STUDENT.set/get_student_addr()`
- `FACULTY.set/get_faculty_addr()`

for the purpose of easy exposition of the next Section 3. In the next two sections, we will fix the loophole we have found with two different methods.

3 VIRTUAL FIELD, AND ITS ENHANCED ACCESSOR RULES

The first method is we will add enhanced accessor rules of the renamed fields to the compiler, and help the programmer to access these fields with disciplines.

Let us examine the flattened view of the fields of each class. Since most Eiffel compilers generate C code as target, let's use C as the target language to make our discussion more concrete.

3.1 Memory layout

The following is the intended memory layout of each class with multiple inheritance and renaming:

```
struct Person {
  char* name;
  char* addr;
};

struct Student {
  char* name;  // from Person
  char* addr;  // from Person
};

struct Faculty {
  char* name;  // from Person
  char* addr;  // from Person
};

struct ResearchAssistant {
  char* name;  // from Person, Student & Faculty (joined)
  char* addr;  // from Person, no renaming
```

```
  char* student_addr;  // from Student and renaming
  char* faculty_addr;  // from Faculty and renaming
};
```

Rule 1 (remove reference identity)**.** *To fix the loophole, first we remove the reference identity relationship among all the renamed fields.*

For example, all the three `*addr` fields in the above example.

When a RESEARCH_ASSISTANT is passed as a PERSON object to a method call or assignment target, and then need to access its `addr` field, depending on its execution context (which we will explain later), this *field access* need to be dispatched to one of:

- `ResearchAssistant.addr`
- `ResearchAssistant.student_addr`
- `ResearchAssistant.faculty_addr`

3.2 Virtual field, and its accessing rules

In traditional OOP languages, we have virtual method dispatch depends on the actual object type, but here the *field access* also needs to be dispatched to the intended renamed new field. Therefore, we would like to introduce the following concept:

Definition 3 (virtual field)**.** If a class field is renamed (anywhere) in the inheritance DAG, we call it a *virtual field*.

For *unplanned* MI, i.e. the programmer can inherit any existing class, and make any necessary feature (here field) adaptations to create a new class, so any field in any class can be a virtual field.

To fix the loophole, we introduce the following enhanced compiler rules:

Rule 2 (virtual field accessor rule)**.**

(1) *the compiler no longer creates any reference identity relationship between the renamed old field and the new field.*

(2) *the programmer must* add new semantic assigning *accessor methods for every virtual field in each class that is immediately below the field's semantic branching site.*

(3) *at each renaming site of a field, the programmer must* override *the new virtual semantic assigning accessor method of that field added in (2), to use the field with the new name.*

(4) *only accessor methods can make direct access (read or write) to those actual fields; while any other methods must use these semantic accessor methods to access those actual fields, instead of accessing those actual fields directly.*

For examples:

- Rule 1.2: PERSON is the semantic branching site of field `addr`, so
  - STUDENT must add new accessors get / set_student_addr()
  - FACULTY must add new accessors get / set_faculty_addr()
- Rule 1.3: RESEARCH_ASSISTANT is the renaming site, so it must override
  - STUDENT.get / set_student_addr() to read / write the renamed field `student_addr`
  - FACULTY.get / set_faculty_addr() to read / write the renamed field `faculty_addr`

Adding new semantic assigning accessors is very important: e.g.

- FACULTY.get_addr() / set_addr() (via PERSON) v.s.
- FACULTY.get_faculty_addr() / set_faculty_addr()

if we only override get_addr() / set_addr() in RESEARCH_ASSISTANT, it will affect both FACULTY and STUDENT class' methods that calls get_addr() / set_addr(), hence still mix the two different semantics; while adding and overriding get_faculty_addr() can establish the semantics of the renamed field FACULTY.addr → RESEARCH_ASSISTANT.faculty_addr.

Furthermore for any other method defined in FACULTY that need the `faculty_addr` semantics of field `addr` (which was only renamed and available in class RESEARCH_ASSISTANT level and down below), it needs to call these new accessor methods instead of the FACULTY.get_addr() / set_addr().

Now let's update class RESEARCH_ASSISTANT to comply with these rules:

Listing 9. research_assistant.e with virtual accessor method override

```
class RESEARCH_ASSISTANT
inherit
        STUDENT rename addr as student_addr      -- field student_addr inherit the dorm semantics
                redefine get_student_addr, set_student_addr
        end
        FACULTY rename addr as faculty_addr      -- field faculty_addr inherit the lab  semantics
                redefine get_faculty_addr, set_faculty_addr
        end
        -- then select, NOTE: not need by SmartEiffel, but needed by GOBO and ISE compiler
        PERSON  select addr end

create {ANY}
   make

feature {ANY}
   get_student_addr():STRING is  do Result := student_addr end  -- override and read the renamed field!
   get_faculty_addr():STRING is  do Result := faculty_addr end  -- override and read the renamed field!
   set_student_addr(a:STRING) is do student_addr := a end   -- override and write to the renamed field!
   set_faculty_addr(a:STRING) is do faculty_addr := a end   -- override and write to the renamed field!

   print_ra() is  -- print out all 3 addresses
     do
       io.put_string(name +" has 3 addresses: <"+ addr +", "+ student_addr +", "+ faculty_addr + ">%N")
     end

   make is
     do
       name := "ResAssis"
       addr := "home"             -- the home semantics
       set_student_addr("dorm")  -- the dorm semantics
       set_faculty_addr("lab")   -- the lab  semantics
     end
end
```

Please pay special attention to the redefined (override) methods get_student_addr(), get_faculty_addr(), set_student_addr() and set_faculty_addr() to see how they implemented the renamed field's *intended* accessor semantics. With these manual overrides, let's run the updated program, and check the new results:

As we noted in Listing 6, the ISE compiler implemented the problematic reference identity semantics of the renamed fields, *since we cannot change their compiler, our method is not applicable to the ISE compiler*. On the other hand, both GOBO and SmartEiffel do not use this reference identity semantics, so we can check their new outputs.

Listing 10. GOBO output: most problems fixed

```
ResAssis has 3 addresses: <home, dorm, lab>
PERSON  size: 24
STUDENT size: 24
FACULTY size: 24
RESEARCH_ASSISTANT size: 40
ResAssis do_benchwork in the: lab
ResAssis take_rest in the: dorm
-- print_student|faculty_addr_direct_field
ResAssis as STUDENT.addr: home
ResAssis as FACULTY.addr: home
-- print_student|faculty_addr_via_accessor
ResAssis as STUDENT.addr: dorm
```

```
13 ResAssis as FACULTY.addr: lab
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test some assignment: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <home, dorm, lab2>
20 ResAssis has 3 addresses: <home, dorm2, lab2>
```

With GOBO Eiffel compiler, most of the problems are fixed, except the two lines 9 & 10 in the middle where the programmer made direct field access (which violates the new programming rules).

Listing 11. SmartEiffel output: most problems fixed

```
1 ResAssis has 3 addresses: <home, dorm, lab>
2 PERSON  size: 12
3 STUDENT size: 12
4 FACULTY size: 12
5 RESEARCH_ASSISTANT size: 20
6 ResAssis do_benchwork in the: lab
7 ResAssis take_rest in the: dorm
8 -- print_student|faculty_addr_direct_field
9 ResAssis as STUDENT.addr: home
10 ResAssis as FACULTY.addr: home
11 -- print_student|faculty_addr_via_accessor
12 ResAssis as STUDENT.addr: dorm
13 ResAssis as FACULTY.addr: lab
14 -- check reference identity
15 ra.addr != ra.faculty_addr
16 ra.addr != ra.student_addr
17 ra.student_addr != ra.faculty_addr
18 -- test some assignment: suppose ra moved both lab2 and dorm2
19 ResAssis has 3 addresses: <home, dorm, lab2>
20 ResAssis has 3 addresses: <home, dorm2, lab2>
```

Again, SmartEiffel's new output is the same as GOBO Eiffel.

Even without direct field access, it's better for a non-accessor method to call only accessor method defined in the *same* class, for example:

Listing 12. problematical call to accessor method from the base class

```
class FACULTY
  ...
  do_benchwork() is
    do
      io.put_string(name + " do_benchwork in the: " + get_addr() + "%N");  -- i.e PERSON.get_addr()
    end
```

If

- FACULTY.do_benchwork() directly call accessor PERSON.get_addr() (instead of FACULTY.get_faculty_addr()), and
- STUDENT.take_rest() directly call PERSON.get_addr() (instead of STUDENT .get_student_addr())

that will be a programming error, since no matter how RESEARCH_ASSISTANT.get_addr() is implemented (or even not overridden at all), there can only be one implementation in RESEARCH_ASSISTANT, so at least one of RESEARCH_ASSISTANT.do_benchwork() and RESEARCH_ASSISTANT.take_rest() will get a wrong address.

Therefore we add the following accessor calling level rule:

Rule 3 (virtual field accessor calling level warning rule – i.e. violations are warnings instead of errors)**.**

(1) *only accessor method can call super-class' accessor method: e.g. FACULTY.get_faculty_addr() call PERSON.get_addr()*

(2) *any non-accessor method can only call virtual field accessor method defined in the* same *class*

But sometimes, the programmers do need to make direct field access or call accessor methods from the base classes, too much such warning messages can be annoying. Therefore we also introduce another syntax for such cases:

Rule 4 (programmer manually verified direct field access or accessor method from the base class). *new feature access operator !:*

- `object!direct_field_access` *or*
- `object!accessor_method_from_base_class()`

to silence the compiler warning messages.

3.3 One step further beyond manual fix

The compiler can be enhanced to issue error / warning messages when detecting these rule violations, and alert the programmer to check and fix them just as we did in our example. While this process works, the programmer needs to re-exam *manually* all the existing code to ensure their semantics are correct.

Eiffel first appeared in 1986, and won the ACM software system awards in 2006, there are many existing users [9] with a possible very big code base. Re-exam and fix all these code base manually is a very complex task, can we do better to avoid this tedious manual approach and make the existing code work *as it is*? The main purpose of MI is to encourage code reuse, we would like to make the method FACULTY.do_benchwork() work correctly according to the programmer's intention *without* adding the extra virtual accessor as we introduced in this section.

Another method is to change the compiler to implement the intended semantics of the rename fields, in the next section we will introduce a new concept called: *rename dispatching based on view stacks* to fix the loophole.

4 RENAME DISPATCHING BASED ON VIEW STACK

(Note: in the following sections, code listings are for language design discussion purpose, hence may not be compilable or executable. The new method we are going to introduce in this section is independent of the previous section; in fact, we assume the previous section, in particular the updated research_assistant.e listing 9 with virtual accessor override does not exist at all. We only assume Eiffel's reference identity semantics of renamed field is *removed*.)

For any new method implemented in class RESEARCH_ASSISTANT (and its descendants) which is related to its FACULTY role, it can use the renamed new field `faculty_addr`, but how about *existing* methods inherited from the super-classes e.g. FACULTY.do_benchwork() method in Listing 3, which accesses the original field via the old name `addr`?

4.1 Virtual field access dispatch

Ideally, when a super-class's method e.g FACULTY.do_benchwork() is called on a RESEARCH_ASSISTANT object, the method needs to access the renamed `addr` field with "lab" semantics, i.e RESEARCH_ASSISTANT.faculty_addr as the programmer introduced in the renaming clause. However in the current Eiffel, it still accesses the old field `PERSON.addr` which has "home" semantics; similarly STUDENT.take_rest() also wrongly access the field

[9] Our own company choose to use Eiffel because of this, and discovered the loophole during our development.

PERSON.addr in RESEARCH_ASSISTANT, whose "home" semantics is not what take_rest() expected "dorm" semantics. So we end up in the situation that these two methods from different super-classes still *share* the same field RESEARCH_ASSISTANT.addr, and this does not achieve feature separation at all.

So after a feature renaming, when an inherited method is called on a derived class object, it still accesses the original feature by the *old* name, which generates problematic semantics different from the programmer's intention by using renaming. The needed dispatch to the features with *new* names is neither discussed in the existing Eiffel language literature, nor implemented by any of the Eiffel compilers that we have tested.

What needed is a semantical dispatch of renamed field, so we introduce the following principle:

Definition 4 (semantical dispatch principle of renamed features)**.** the purpose of feature renaming is to resolve feature name clash while achieving the programmer's renaming intention; when the original feature (by the old name) is accessed (both read and write) on a sub-class object in the super-class' method, that feature access needs to be dispatched to the renamed feature (by the new name) in the sub-class.

This renamed feature dispatching semantics is different from Eiffel's original reference identity semantics: e.g. with reference identity semantics, all these three notation

(1) RESEARCH_ASSISTANT.addr
(2) RESEARCH_ASSISTANT.student_addr
(3) RESEARCH_ASSISTANT.faculty_addr

refer to the same field; while with renamed feature dispatching semantics these three are *separated* fields (with different physical memory locations), and for any method call or statement that need access to the addr field, if the execution context is in the:

(1) PERSON branch, it needs to be dispatched to RESEARCH_ASSISTANT.addr
(2) STUDENT branch, it needs to be dispatched to RESEARCH_ASSISTANT.student_addr
(3) FACULTY branch, it needs to be dispatched to RESEARCH_ASSISTANT.faculty_addr

4.2 Field access execution context is object view stack dependent

In this subsection we will show that field access execution context is object view stack dependent.

4.2.1 execution context: call stack.

Let's exam the execution context when STUDENT.set_student_addr() is called on a RESEARCH_ASSISTANT object: the method is defined in class STUDENT.e in Listing 2, which in turn calls PERSON.set_addr() method, and the final assignment statement there writes to the addr field, so the call stack at the assignment site is (from the top to the bottom)[10]:

Listing 13. the actual call stack of ra.set_student_addr()

```
set_addr()          --  PERSON.e line 9  with Current type: PERSON
set_student_addr()  -- STUDENT.e line 6, with Current type: STUDENT
ra.set_student_addr("dorm")          -- with Current type: RESEARCH_ASSISTANT
```

As we have just discussed, this write needs to be performed on the student_addr field of RESEARCH_ASSISTANT (please refer to the renaming DAG of Fig 2) hence[11]:

[10] The Eiffel keyword Current is just like this in C++ & Java, or self in Python, which represents the current object instance.
[11] Here we use "==" to mean reference identity testing (as in C++/Java).

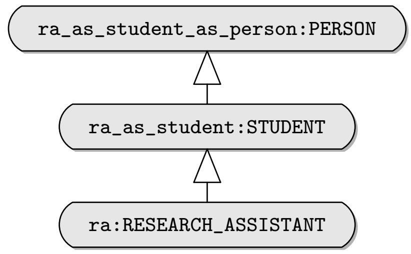

Fig. 3. view stack as a set of "lenses" on the object ra

Listing 14. write to the renamed field

```
1 ra.set_student_addr("dorm");        -- write to the STUDENT.addr field
2 assert(ra.student_addr == "dorm");  -- read the renamed student_addr field
3
4 ra.set_faculty_addr("lab");         -- write to the FACULTY.addr field
5 assert(ra.faculty_addr == "lab");   -- read the renamed faculty_addr field
```

For line 1: the actual assignment statement to the field `(PERSON.)addr` is in the method `PERSON.set_addr()`, and at that assignment site (i.e. line 9 of PERSON.e of Listing 1), the call stack of the `Current` object's type from bottom to the top is:

$$callStackTypes = [RESEARCH\_ASSISTANT, STUDENT, PERSON]$$

For line 4: similarly the call stack is:

$$callStackTypes = [RESEARCH\_ASSISTANT, FACULTY, PERSON]$$

Note the *same* (by-name) field `PERSON.addr` is modified by the *same* method `PERSON.set_addr()`, and is invoked on the *same* object (`ra`), but the actual assignments need to be made on two *different* actual fields: `ra.student_addr` and `ra.faculty_addr`, due to the different call stacks.

Conversely, to read a renamed field e.g. ra.get_student_addr(), the actual call stack is:

Listing 15. the actual *call stack* of ra.get_student_addr()

```
get_addr()           --  PERSON.e line 8  with Current type: PERSON
get_student_addr()   -- STUDENT.e line 5, with Current type: STUDENT
ra.get_student_addr()                   -- with Current type: RESEARCH_ASSISTANT
```

and the *actual* field which is needed to be read here is `ra.student_addr`.

4.2.2 *execution context: view stack.*

Also, let's consider the following assignment statements sequence:

Listing 16. assignment chain, and *view stack*

```
1 ra: RESEARCH_ASSISTANT;
2 ra_as_student: STUDENT := ra;
3 ra_as_student_as_person: PERSON := ra_as_student;
4 ra_as_student_as_person.addr := "dorm";
```

At the last line 4, the actual assignment site, there is no method call stack; however, comparing with the previous code listing 13 of the *call stack* there, the variable `ra_as_student_as_person` here has a *view stack* of the object `ra` it holds:

$$viewStackTypes = [RESEARCH\_ASSISTANT, STUDENT, PERSON]$$

Definition 5 (object view stack)**.** Here we generalize the concept of method call stack to *object view stack*, at the bottom of the stack is the object's actual type RESEARCH_ASSISTANT, and the `ra` object is assigned to variable `ra_as_student` of type STUDENT, and then `ra_as_student_as_person` of type PERSON. At this point, the object is used (viewed) as if it's a PERSON type thru a stack of lenses all the way down to the actual object of type RESEARCH_ASSISTANT.

Example 4.1. In the following, the same `RESEARCH_ASSISTANT ra` object is held by different variables of its super-class type, hence has different view stacks:

Listing 17. different views of the same object, direct field access

```
1  ra: RESEARCH_ASSISTANT  -- all the variables refer to this same RESEARCH_ASSISTANT object
2  ra_as_student: STUDENT := ra;
3  ra_as_faculty: FACULTY := ra;
4  person: PERSON := ra;
5
6  -- accessing the field by the same name does not mean accessing the same field!
7  assert(ra_as_student.addr == "dorm");  -- [RESEARCH_ASSISTANT, STUDENT] view of ra object
8  assert(ra_as_faculty.addr == "lab" );  -- [RESEARCH_ASSISTANT, FACULTY] view of ra object
9  assert(person.addr == "home");         -- [RESEARCH_ASSISTANT, PERSON]  view of ra object
10
11 -- view stack of assignment chain
12 person := ra_as_student;               -- assign ra to PERSON via STUDENT
13 assert(person.addr == "dorm");         -- [RESEARCH_ASSISTANT, STUDENT, PERSON] view of ra object
14
15 person := ra_as_faculty;               -- assign ra to PERSON via FACULTY
16 assert(person.addr == "lab" );         -- [RESEARCH_ASSISTANT, FACULTY, PERSON] view of ra object
```

note on line 13, assert(person.addr == "dorm"); while on line 16, assert(person.addr == "lab"). This shows view stack is different from usual method call stack, i.e. even in the same scope assigning to a local variable of a different type will change the view stack of the object.

Summary: every access (write and read) of a renamed field needs to be dispatched based on its renaming DAG, *and* the view stacks of the variable at the access site. To the best of the author's knowledge, this behavior has never been documented in any previous OOP literature. We call it *rename dispatching based on the view stack.*

Example 4.2. As a consequence of such renamed field dispatch, now the following accessor method calls will return the intended renamed field:

Listing 18. different views of the same object, accessor method call

```
ra_as_student.get_addr();  -- now return "dorm"
ra_as_faculty.get_addr();  -- now return "lab"
       person.get_addr();  -- now return "home"
```

which means we can make the existing code work as it is by adding rename dispatching to the compiler.

4.3 Variable as object and view stack holder

An object can be hold by different variables, and each variable hold a different view (history) of the object. For example, a same RESEARCH_ASSISTANT object can be passed to both the methods call do_benchwork() and take_rest() simultaneously, e.g on two different threads. So the view stack cannot be hold by the object itself (then will be shared by two different threads); instead each execution context need to maintain its own separate view stack of the object.

Definition 6. Each variable is a "fat pointer", which has two components:

(1) the actual object

(2) the view stack of the object

And we'd introduce the following rules to update view stacks:

Rule 5 (view stack updating rule). *Note, in the following rules,* variables *also include compiler generated temporary variables (not visible by the programmer)*

(1) *var:V = new O(), when an object of type O is created and assigned to var of type V, then*

$$view\_stack(var) = [O, V]$$

(2) *var:V = u, when a variable u is assigned to a variable of type V, then*

$$view\_stack(var) = view\_stack(u) + [V]$$

i.e, push type V on to the top of the view stack.

(3) *function(var:V) be called with function(u), when a variable u is passed to a function as a parameter of type V, then*

$$view\_stack(var) = view\_stack(u) + [V]$$

i.e. push type V on to the top of the view stack.

Example 4.3 (assignment chain). `ra` is first assigned to `ra_as_student`, and then to `person`:

$$view\_stack(person) = [RESEARCH\_ASSISTANT, STUDENT, PERSON]$$

4.4 rename dispatching

Each variable carries a type stack, which holds the type information of the object's view history.

Rule 6 (Virtual field dispatching rule). *Given an object's view stack S, find the shortest path P in the object's field's renaming DAG from the top(S) to the bottom(S), such that* $S \subseteq reverse(P)$,

(1) *if there is only one shortest path, dispatch to the field's final rename in the renaming DAG.*

(2) *if there are multiple such shortest paths, raise a run-time exception.*

Now let us review our previous write and read access to the virtual field `addr`:

Example 4.4.

(1) for Listing 13, at the assignment site (line 9 of class PERSON in Listing 1):

$$view\_stack(Current) = [RESEARCH\_ASSISTANT, STUDENT, PERSON]$$

we can find the corresponding path in Fig 2 of the renaming DAG of field `addr` is:

$$PERSON \rightarrow STUDENT \rightarrow RESEARCH\_ASSISTANT$$

and the final name of the field is `student_addr`, so the assignment of string "dorm" in set_addr() to virtual field `addr` will be made on the actual field `ra.student_addr`.

(2) and for line 7 of Listing 17,

$$view\_stack(ra\_as\_student) = [RESEARCH\_ASSISTANT, STUDENT]$$

we can find the corresponding path in Fig 2 of the renaming DAG of field `addr` is:

$$STUDENT \rightarrow RESEARCH\_ASSISTANT$$

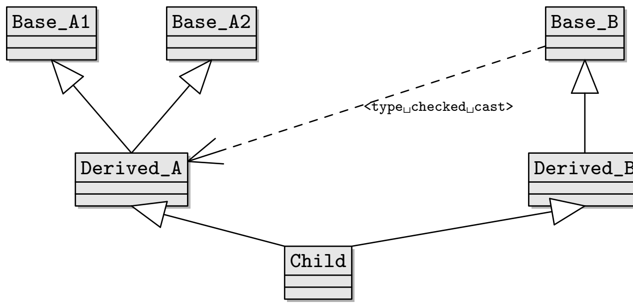

Fig. 4. multiple inheritance branches, and type cast from Base_B to Derived_A

and the final name of the field is `student_addr`, so the field access of virtual field `addr` will be dispatched to the actual field `ra.student_addr`, thus the assertion holds:

```
assert(ra_as_student.addr == "dorm");  --  [RESEARCH_ASSISTANT, STUDENT] view of ra object
```

4.5 Normalized view stacks, and dispatch optimization

With multiple inheritance, an object's view stack can be more complex than just along a single linear branch, let's consider the following MI DAG: Suppose there is an object of type Child, this object can be assigned (with compiler generated type checking) to a reference variable of any of its base class type, and in any sequence. In Eiffel this is called assignment attempt, and in other languages (e.g. C++ / Java) it is called (type checked) cast. For example, consider the following assignment attempts:

Listing 19. type cast

```
1 child: Child
2 base_b: Base_B
3 derived_a: Derived_A
4
5 base_b := child
6 derived_a ?= base_b   -- type cast
7 if derived_a /= Void then
8   derived_a.some_method_call()
9 end
```

all the assignment attempts will succeed, so the view_stack(derived_a) at line 8 is [Child, Base_B, Derived_A], and in particular Base_B can be totally unrelated to Derived_A. This example shows in MI, it is possible to cast to a variable of type A which is on a *different* branch in the inheritance DAG from the object's current holding variable's declaration type B. To simplify view stack management, let's define the *normalized* view stack:

Definition 7 (Normalized view stack)**.** Let view stack

$$S = [T_0, T_1, \dots T_{n-1}]$$

and T_0 be the object's actual type at the bottom of the stack, we say S is a normalized stack, if for any $j > i$, T_j is a superclass of T_i. Or in short, all the types in a normalized view stack needs to be in the strict monotonic super-classing total order.

To normalize an arbitrary view stack, we introduce the following rules:

Rule 7. *If two adjacent elements of the view stack are the same [A, A], normalize it to [A].*

Rule 8 (type cast)**.** *Suppose [A, B] are the top two elements of a view stack, i.e. VS + [A, B], then update the view stack:*

(1) *first cast to the* greatest common derived class *gcd(A, B) of A and B: VS + [gcd(A, B)]*
(2) *then cast from gcd(A, B) to B: VS + [gcd(A, B) + B]*

Example 4.5. normalize view stack [RA, FACULTY, STUDENT]:

(1) $[RA, RA]$, since RA is the gcd(FACULTY, STUDENT)
(2) $[RA, RA, STUDENT]$
(3) $[RA, STUDENT]$

Example 4.6. cast up and down along the same branch: normalize view stack [RA, FACULTY, RA]

(1) $[RA, RA]$, since RA is the gcd(FACULTY, RA)
(2) $[RA]$

The normalized view stack can only effectively grow in the direction to the root class of the inheritance DAG.

Theorem 4.7 (limited view stack length). *The max view stack length is the max depth of the inheritance DAG.*

The most straightforward implementation of the dispatch method is at least linear to the stack length; to speed up, we can create a hash-table by enumerating all the possible type view stacks during the compilation, and reduce the runtime dispatching cost to O(1) (since both the full inheritance and field renaming DAG are known at compile time, the compiler can create a perfect hash-table).

5 IMPLEMENTATION: VIRTUAL FIELD DISPATCH BASED ON VIEW STACKS

Due to paper length limit, we will only give a very brief description of our implementation. For demo purpose, our implementation sets a few restrictions (e.g. the max view stack length to be 8), and is not optimized (except for the virtual field dispatch using a hash-table). All the source code is available on github.

Since there are fewer Eiffel developer utility tools available compared with other more popular languages used in the industry, for easy experiment and quick verification purpose, we choose to use Python tools to implement the ideas we discussed in the previous section. We add Eiffel style renaming syntax to a Python-like language and generate target code in D. This is not so much different from the traditional practice that most Eiffel compilers compile Eiffel language to target code in C. With enough time and resource permitting, the ideas presented in this paper can be implemented in any programming language.

We added the following renaming syntax to Python:

```
class ResearchAssistant(
    Student(rename _addr as _student_addr),
    Faculty(rename _addr as _faculty_addr),
    Person):
...
```

In the generated target code in D:

(1) the class body mostly defines the fields memory layout and some helper methods.

(2) we move class method out of class: e.g a method call `self.foo(args)` becomes `foo(self, args)`[12]
(3) field accessors are implemented as functions.

For example:

Listing 20. demo.yi

```
class Faculty(Person()):  # i.e Faculty inherit Person
  def do_benchwork(self):
    ...
```

Listing 21. generated demo.d: move class method out of class definition body

```
void do_benchwork(FACULTY self) {  // class method implementation
  ...
}

ref string Person_addr(Person self) {  // field accessor implementation
  ...
}
```

Each class has an internally assigned type id, and in this demo, we restrict max type id < 256 (i.e 1 byte), and the max length of the hash of view stack to be 8 bytes, so the max view stack length <= 8, as the following the pseudo target code shows:

```
__Person.___typeid = 0x00;
__Student.___typeid = 0x01;
__Faculty.___typeid = 0x02;
__ResearchAssistant.___typeid = 0x03;

alias TypeStackPathHashT = ulong;  // 8 bytes
```

In our simple demo implementation, the objectViewStackHash actually carries the whole view stack: view stack push() / pop() are simulated by bits-shifting, and the highest byte represents the bottom of the view stack. For example, the generated dispatch hash-table for Person.addr is:

```
ref string Person_addr(Person self) { // virtual field dispatch table (VFDT)
  self = cast(Person)(self.cloneH());
  self.pushObjectViewStack(0);
  switch (self.objectViewStackHash) {
  case          0x030200:  return (cast(ResearchAssistant)self).__Faculty_addr;  // renamedCount=1
  case            0x0300:  return (cast(ResearchAssistant)self).__addr;  // renamedCount=0
  case            0x0100:  return (cast(Student)self).__addr;  // renamedCount=0
  case          0x030100:  return (cast(ResearchAssistant)self).__Student_addr;  // renamedCount=1
  case            0x0200:  return (cast(Faculty)self).__addr;  // renamedCount=0

  case 0:  {return self.__addr;};
  default:  enforce(false, format("%x", self.objectViewStackHash));
  }
 {return self.__addr;}
}
```

here 0x030200 represents the view stack [0x03, 0x02, 0x00], i.e [ResearchAssistant, Faculty, Person], and the actual field dispatched to is: `(cast(ResearchAssistant)self).__Faculty_addr`, which is what the programmer intended.

The full code of demo.yi, which is equivalent of the Eiffel code of Section 2, is in Appendix A.1, and the outputs are:

Listing 22. demo.yi output

```
ResAsst has 3 addresses: <home dorm lab>
ResAsst do_benchwork in the: lab
ResAsst take_rest in the: dorm
-- print_student|faculty_addr_direct_field
```

[12]Define methods out of class body has another benefits: we can use multi-methods dispatch, i.e virtual dispatch on all the method's args, instead of the single (implicit) current object `this / self`.

```
ResAsst as STUDENT.addr: dorm, age=18
ResAsst as FACULTY.addr: lab, age=18
-- print_student|faculty_addr_via_accessor
ResAsst as STUDENT.addr: dorm
ResAsst as FACULTY.addr: lab
-- test some assignment: suppose ra moved both lab2 and dorm2
ResAsst has 3 addresses: <home dorm lab2>
ResAsst has 3 addresses: <home dorm2 lab2>
```

As we can see, the results are all what the programmer has expected now. We also demonstrate *on purpose* that two distinct fields `Student._student_age` and `Faculty._faculty_age` are joined into one single field `ResearchAssistant._age`, which the three Eiffel compilers all failed to join with the message: "Error: two or more features have the same name (age)."

5.1 Legacy Eiffel code migration plan: combine the two methods

As we can see in this second method, every field access (except the raw access operator with "!") becomes a method call, and for every variable assignment (including parameter passing in method call) the language runtime needs to maintain the object view stacks. These operations will increase both the memory and runtime overhead of the compiled program. While the first method we introduced in the previous Section 3 is more efficient at runtime.

Actually to migrate legacy Eiffel code, we can combine these two methods:

(1) first, use the second method of this section enhanced compiler to generate test cases for the intended semantics.
(2) then, auto generate new and override semantic assigning accessor methods according to Rule 2 & 3 for the corresponding virtual fields, and use the first method enhanced compiler to validate these test cases,

6 DISCUSSIONS

6.1 Renamed methods

In Eiffel, methods can also be renamed, and the renamed methods need to be treated in the same way as renamed fields.

In 2015, an Eiffel programmer found and raised a similar question regarding renamed methods on the web: https://stackoverflow.com/questions/32498860/

> I am struggling to understand the interplay of multiple inheritance with replication and polymorphism. Please consider the following classes forming a classical diamond pattern.

```
deferred class A
  feature
    a  deferred end
  end

deferred class B
  inherit A
    rename a as b end
  end

deferred class C
  inherit A
    rename a as c end
  end

class D
  inherit
    B
    C select c end
feature
  b do print("b") end
  c do print("c") end
```

```
end
```

> If I attach an instance of D to an object `ob_as_c` of type C, then `ob_as_c.c` prints "c" as expected. However, if attach the instance to an object `ob_as_b` of type B, then `ob_as_b.b` will also print "c".
> Is this intended behavior? Obviously, I would like `ob_as_b.b` to print "b".

Both Bertrand Meyer and Emmanuel Stapf (the lead developer of EiffelStudio) replied to that question, but they only mentioned that in (virtual) dynamic binding, there is only one version of the method on D, can be called.

Instead we think, first, this is another problem in Eiffel's `select` clause as we have mentioned earlier: after the two feature renamings (separation) there is *no* more name clash on `a`, however they are forced to be joined again (by the `inherit C select c`), i.e `ob_as_b.b` actually calls `d.c`. As the programmer who asked question finally put it: "Unfortunately, this makes Eiffel's inheritance features much less useful to me."

Second, just same as our treatment of virtual fields: the compiler should keep methods `D.b & D.c` separate, and the `select` clause is not needed. Now dispatch `ob_as_b.b`, `ob_as_c.c` and even `ob_as_a.a` by rename DAG and view stack first, and then dispatch as virtual method (if there is any further override) as we did in Section 4. This will achieve what the user wanted in the original question.

6.2 Repeated inheritance

With our treatment of virtual field, repeated inheritance is not allowed, neither the virtual field nor the view stack introduced in this paper can handle repeated inheritance directly. However, this can be supported by the following simple rewriting:

```
class B inherit A, A   -- repeatedly inherit A multiple times by the same sub-class is not allowed!

-- but, we can rewrite to
class A1 inherit A

class B inherit A, A1  -- this is fine
```

6.3 Compare virtual method dispatch and rename dispatching

Virtual *field* dispatch is different from virtual *method* dispatch in that virtual field dispatch depends on *the whole view stack* from the holding variable type down to the actual object type; while virtual method dispatch depends on *only* the actual object type.

6.4 Future works

6.4.1 In the theoretical aspects: study the interplay of virtual fields with other language constructs. In OOP virtual method is a well studied concept, while we have not found any discussion of virtual field. The interplay of virtual field with other constructs of the OOP language is yet to be explored for new opportunities (or new problems).

6.4.2 In the practical aspects: design more efficient implementations. Design more efficient management strategy to reduce view stack memory requirement and dispatch runtime overhead.

7 RELATED WORK

There is a long history of inheritance and polymorphism have been studied since 1980s, and there are copious literature on multiple inheritance, e.g. [Compagnoni and Pierce. 1993], [Cardelli 1988], [van Limberghen and Mens 1996], [Taivalsaari 1996], [Bracha and Cook 1990].

In the following we will only compare some of the earlier work that has some similarity as our work, and highlight the difference.

In [Carré and Geib 1990] Carré and Geib introduced the point-of-view (PoV) notion of multiple inheritance. While both their work and our work try to solve the same problem of dynamic dispatching of class fields, our work differs from their PoV in the following ways:

(1) Our view stack approach dynamically tracks the *full object-to-target runtime path*, while their approach only considers the <source object, target type> *two points pair*.
(2) In their paper, they did not give the detailed rules on *how* to do selection (i.e dispatch), especially if there are *multiple paths* between PoV<C, O>, (and its even not clear if they handle such cases at all); while we give very specific Rule 5 on how to dispatch, even for *multiple paths*.
(3) Their approach only uses the *global* inheritance lattice for *all* the Pov(C, O) calculation; while in our approach, *each renamed field has its own renaming DAG* (which can be different from the global inheritance DAG), so in our stack view dispatch calculation we use each fields own renaming DAG (and for never-renamed field, e.g. `name`, there is no need for dispatch).
(4) In their approach, all inherited fields are separated; while in our approach fields can be shared or separated (renamed) according to the programmers application semantics.

In [Chambers et al. 1982], the sender path tiebreaker rule of SELF can only handle the case where if "only one of the parents is an ancestor or descendant of the object containing the method that is sending the message (the sending method holder)", which means this rule cannot handle the inheritance relationship in the diamond problem, while our approach can handle it.

In [Borning and Ingalls 1982], Borning and Ingalls discussed the multiple inheritance in Smalltalk-80. They also considered when the same method is inherited via several paths, an error will be reported to the user. In their paper, they did not give the detail of the method resolution method that they used. While our approach gives every specific rules on how to do runtime dispatch, and when to raise exceptions.

In [Burow et al. 2018], an orthogonal policy Object Type Integrity (OTI) is proposed to dynamically track C++ object types. So instead of allowing a set of targets for each dynamic dispatch on an object, only the single, correct target for the objects type is allowed. Their work is based on C++ class model where there is no feature renaming, while our work is toward fixing Eiffel's renaming mechanism.

In [Ngomo 2021], Ngomo described a non-linear and non-deterministic approach of the semantics of multiple inheritance in their problem domain, i.e. multiple specification of logical objects. While our approach is deterministic, so the programmers can easily reason about the programs.

In [Suchánek and Pergl 2020], Suchánek and Pergl studied the method resolution order in Python, and showed that inheritance can be implemented with minimisation of combinatorial effect using the patterns.

8 CONCLUSIONS

We found the reference identity semantics of Eiffel's field renaming mechanism is problematic, as demonstrated in the diamond problem of MI. We introduced a new concept called virtual field and proposed two methods to solve it:

(1) always add new and use semantic assigning accessors, avoid direct field access.

(2) rename dispatching based on view stack.

However, the second method will have some negative impact on the performance of the resulting program, and cause increased complexity of compiler implementation. Further research on compiler optimization is needed.

A APPENDIX

A.1 Demo.yi

All the source code of this paper can be found in the supplementary material: eiffel_rename.tgz.

Listing 23. demo.yi

```
LAB:str = "lab"
HOME:str = "home"
DORM:str = "dorm"


class Person(object):

  _name:str
  _addr:str

  def __init__(self):
    self._name = ""
    self._addr = ""

  def get_addr(self) -> str:
    r:str = self._addr
    return r;


class Student(Person()):

  _student_age:int  # distinct field on purpose

  def __init__(self):
    self._addr = DORM

  def get_student_addr(self) -> str:
    r:str = self.get_addr()
    return r

  def take_rest(self) -> str:
    print(self._name)
    print(" take_rest in the: ")
    print(self._addr)
    print("\n")
    return self._addr


class Faculty(Person()):

  _faculty_age:int  # distinct field on purpose

  def __init__(self):
    self._addr = LAB

  def get_faculty_addr(self) -> str:
    r:str = self.get_addr()
    return r

  def do_benchwork(self) -> str:
    print(self._name)
    print(" do_benchwork in the: ")
    print(self._addr)
    print("\n")
    return self._addr


class ResearchAssistant(
    Student(rename _addr as _student_addr, rename _student_age as _age),
    Faculty(rename _addr as _faculty_addr, rename _faculty_age as _age),
    Person):

```

```
  def print_ra(self):
    print(self._name)
    print(" has 3 addresses: <")
    print(self._addr)
    print(" ")
    print(self._student_addr)
    print(" ")
    print(self._faculty_addr)
    print(">\n")


def print_student_direct_field(s:Student):
  print(s._name)
  print(" as STUDENT.addr: ")
  print(s._addr)  # output "dorm"
  print(", age=")
  print(s._student_age)
  print("\n")
  assert(s.get_addr() == DORM)


def print_faculty_direct_field(f:Faculty):
  print(f._name)
  print(" as FACULTY.addr: ")
  print(f._addr)  # output "lab"
  print(", age=")
  print(f._faculty_age)
  print("\n")
  assert(f.get_addr() == LAB)


def print_student_addr_via_accessor(u:Student):
  r:str = u.get_student_addr()
  print(u._name)
  print(" as STUDENT.addr: ")
  print(r)
  print("\n")

def print_faculty_addr_via_accessor(u:Faculty):
  r:str = u.get_faculty_addr()
  print(u._name)
  print(" as FACULTY.addr: ")
  print(r)
  print("\n")


def test_Faculty():
  f1:Faculty = Faculty()
  f1._name = "Faculty"
  f1._addr = LAB
  assert(f1.get_addr() == LAB)
  assert(f1.do_benchwork() == LAB)


def main():
  ra:ResearchAssistant = ResearchAssistant()
  ra._name = "ResAsst"
  ra._age = 18
  ra._addr = HOME
  ra._student_addr = DORM
  ra._faculty_addr = LAB
  ra.print_ra()

  # suppose the same `ra` object is passed as Faculty do_benchwork(), and Student take_rest() in parallel
  # or same object passed as two different (type) args to a same func
  # each method need to take its view of the
  assert(ra.do_benchwork() == LAB)
  assert(ra.take_rest()    == DORM)

  print("-- print_student|faculty_addr_direct_field\n")
  print_student_direct_field(ra)
  print_faculty_direct_field(ra)

  print("-- print_student|faculty_addr_via_accessor\n")
  print_student_addr_via_accessor(ra)
  print_faculty_addr_via_accessor(ra)

  print("-- test some assignment: suppose ra moved both lab2 and dorm2\n")
  ra._faculty_addr = "lab2"
  ra.print_ra()
  ra._student_addr = "dorm2"
```

```
ra.print_ra()
```

REFERENCES

Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. *OOPSLA '96 Conference Proceedings. ACM Press.* (1996), 6982.

Alan H. Borning and Daniel H. H. Ingalls. 1982. Multiple inheritance in smalltalk-80. *Proceedings of the Second AAAI Conference on Artificial Intelligence (AAAI'82). AAAI Press* (1982), 234237.

Gilad Bracha and William Cook. 1990. Mixin-based inheritance. *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (1990), 303311.

Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. Cfixx: Object type integrity for c++ virtual dispatch. In *Symposium on Network and Distributed System Security (NDSS)*.

Luca Cardelli. 1988. A semantics of multiple inheritance. *Inf. Comput. 76.2/3* (1988), 138–164.

Bernard Carré and Jean-Marc Geib. 1990. The point of view notion for multiple inheritance. *ACM Sigplan Notices 25.10* (1990), 312–321. https://dl.acm.org/doi/pdf/10.1145/97946.97983

Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. 1982. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *LISP and Symbolic Computation 4* (1982), 207–222.

Adriana B. Compagnoni and Benjamin C. Pierce. 1993. Multiple inheritance via intersection types. *Computing Science Institute, Department of Informatics, Faculty of Mathematics and Informatics, [Katholieke Universiteit Nijmegen]* (1993).

ECMA. 2006. *Eiffel: Analysis, Design and Programming Language*. ECMA International. https://www.ecma-international.org/wp-content/uploads/ECMA-367_2nd_edition_june_2006.pdf

Bertrand Meyer. 1997. *Object-oriented Software Construction, 2nd Ed.* Prentice Hall. https://bertrandmeyer.com/OOSC2/

Macaire Ngomo. 2021. MULTIPLE INHERITANCE MECHANISMS IN LOGIC OBJECTS APPROACH BASED ON A MULTIPLE SPECIALISATION OF OBJECTS. *International Journal of Computer Trends and Technology, vol. 69, no. 10, pp. 1-11, 2021* (2021). https://doi.org/10.14445/22312803/IJCTT-V69I10P101

Bjarne Stroustrup. 1991. *The C++ Programming Language (Second Edition)*. Addison-Wesley.

Marek Suchánek and Robert Pergl. 2020. Evolvability Analysis of Multiple Inheritance and Method Resolution Order in Python. *PATTERNS 2020 : The Twelfth International Conference on Pervasive Patterns and Applications* 8 (2020), 11.

Antero Taivalsaari. 1996. On the notion of inheritance. *ACM Comput. Surv. 28, 3 (Sept. 1996)* (1996), 438479. https://doi.org/10.1145/243439.243441

Marc van Limberghen and Tom Mens. 1996. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Syst. 3.1* (1996), 1–30.

Guido van Rossum. June 23, 2010. *The History of Python: Method Resolution Order*. https://python-history.blogspot.com/2010/06/method-resolution-order.html