SUPPLEMENTARY MATERIAL FOR: INTRODUCING VIRTUAL FIELD AND RENAME DISPATCHING BASED ON VIEW STACK TO FIX EIFFELS RENAME LOOPHOLE

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.

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.

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)`[1]
(3) field accessors are implemented as functions.

For example:

Listing 1. demo.yi

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

Listing 2. 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
```

[1]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`.

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.