CS131 Compilers Project 3 Code Review

Shuyue WANG 2018533070

Yining SHE 2018533219

Compilation errors

In this part, we will share some trouble we met during coding and how we solved them. Most of compilation errors are solved by c++ extensions on vscode so they didn't bother us too much. Other errors were fixed by running the provided `semant` and our semant at the same time and compared ours with the standard one.

Infinite loop

In tag **Infinite_loop**, we fixed the bug of infinite loop. We detected it using a simple example:

Three classes A, C, D. A inherits C, C inherits A, D inherits C. Now we want to find D's parent, then dead lock happens.

We couldn't detect loop inheritance like this before. This problem was solved by introducing `std::set<Symbol> already_parent` to record the parents already visited.

SELF_TYPE

In tags **SELF_TYPE_1**, **SELF_TYPE_2**, **SELF_TYPE_3**, we fixed bugs introduced by `SELF_TYPE`. First, we realized that if a symbol is `SELF_TYPE`, it does not need to occur in `class_table`, so we added some judgment conditions before traversing `class_table`. Second, the utility functions `bool ClassTable::is_inherit()` and `Symbol ClassTable::first_common_anc()` could not do well with `SELF_TYPE`.

Segment Fault

In tag **Segment_Fault**, we solved a segmentation fault by properly using the `for` loop, which is a silly bug.

Neat Code

What we did

1. Add comments to make codes more readable.
2. Control the length of code in one line by breaking sentences in the right place to increase code readability.
3. Unified naming variables.

What we need

1. Change the pointer to smart pointer to save the residual code.
2. Delete codes for debugging. Or use a globle bool variable `debug` to control the on/off of debug_mode. Add `if(debug)` when we want to log the debug message.
3. Many of codes handling `SELF_TYPE`, espacially those in `if` clause, were added to fix bugs introduced by `SELF_TYPE`. Maybe we can solve them in a more elegant way by adding util functions.

Design pattern

Here, we mainly talk about our design pattern of type checking.

We added declaration of function `check_type` in each class. `check_type` is a method that exists in all kinds of `class_ tree` nodes. In this way, we don't need to know which kinds of class the tree node is before we do type checking, but call `check_type` directly.
Therefore, `check_type` is like a kind of unified external api. It is **Facade Pattern**, to make a complex subsystem easier to use, a simple interface should be provided for a set of interfaces in the subsystem. `check_type()` is the interface.

Memory Control

What we did

We didn't do memory control in this project, though we have used `new` to install new symbols to `attr_table`, which will cause memory leak.

What we need

We should do **garbage collection**. There are two ways:

1. `delete` the memory we have allocated before when the program exits, no matter it exits w/o errors. We can put the `delete` part into a function `void garbage_collection()` and call it in the last of function `void program_class::semant()`'s body as well as anywhere `exit(1)` exists. As our `new` is only called to install new symbols to `attr_table`, we can just traverse `attr_table` and `delete` its members one-by-one.
2. Use `std::auto_ptr` at the beginning instead of using `new`. Since mordern C++ will do garbage collection automatically with smart pointers, why not use it to avoid niusances?

Misc

The difference of map and unordered_map

std::map is implemented with a balanced binary search tree (e.g. Red–black tree). So each pair in std::map is stored in order based on its key value.
std::unorder_map is implemented with a hash table, which is unordered and takes more space to save.

Why vector but not list

We found that the element in vector can be accessed more directly using index.

How to get all ancestors

First, at the beginning of `sement()`, we would check whether there is something wrong related with inheritance. If there is, we would halt. So in this case, it not possible to call `get_all_anc()`. Therefore, we suppose, there is no inheritance error when we call `get_all_anc()`.
When implementing `get_all_anc()`, we simply use a while loop to go over its ancestors. We start from the input class, and at each step, we go to the parent of the current class until there is no more parent.

Explain the semantics if you write `for(auto& iter: class_table)`

`for(auto& iter: class_table)` is using `iter` to get the value of each element of `class_table`. Also, since `&` is used here, `iter` is a reference, which means that we change directly change the element in `class_table` by changing `iter`.

Justify the process of traverse the features and BFS or probe. Compare the pros and cons of them.

Here, we intend to check whether any class's method overwrites its ancestor's method with a different formal type or with different number of formals.
We first have an outer loop that goes through every class.
Then, at each step, we have an internal loop that goes through every feature in current class.
Next, we get current class's all ancestors, and compare current method's with their method with the same name.
We don't know what you mean by "BFS or probe. Compare the pros and cons of them", so we just explain our idea here to you.

Why requires virtual function `=0`? Use godbolt to explain with and without `=0`?

A **pure virtual function** is declared by assigning 0 in the declaration. Without `=0`, it's just a virtual function without pure. The pure virtual function makes the class an **abstract class**.
Here, `Feature_class` is just an abstract class, we don't want to instantiate it under any circumstances. So we add `=0` to force its child classes to implement this function.
We test it on https://godbolt.org/ using piece of code:

```
class Symbol{};

class Class__class{
public:
   virtual Symbol get_name() = 0; /* We edit this line to compare. */
};

class class__class : public Class__class {
protected:
   Symbol name;

public:
   Symbol get_name() {
      return name;
   }
};

int Main(){
    class__class aaa;
    Symbol ab = aaa.get_name();
    return 0;
}
```

With `=0`, the complier codes will be more than without `=0`. The additional codes are as **below** using https://godbolt.org/.
We can see that if we do not add `=0`, there will be no assembly code for class `Class__class` just like it never occurs. Because without `=0`, it's just a normal virtual function, we do not need to

implement it after and it is useless. As a result, the useless function as well as the class it in is ignored by compiler.
With `=0`, the pure virtual function must be implemented later in derived class, otherwise the derived class will also become abstract class. So the function implemented in child class must override the pure virtual function, the original pure virtual function is meaningful in this case.

```
typeinfo for Class__class:
        .quad   vtable for __cxxabiv1::__class_type_info+16
        .quad   typeinfo name for Class__class
typeinfo name for Class__class:
        .string "12Class__class"
```

How we solve the dead code concerning `SELF_TYPE` and others.

By running some example codes using provided `semant` and comparing ours with theirs.

1. We found that if `return_type` is `SELF_TYPE`, `expr_type` should also be `SELF_TYPE`, which we have not considered before.
2. Fixed some error message to match the standard one.

Line number problems

Line number problems are introduced by using

```
ostream& ClassTable::semant_error(Class_ c)
{
    return semant_error(c->get_filename(),c);
}
```

This is not an accurate way to report errors, since it doesn't include `tree_node` information. If we invoke `semant_error(Symbol filename, tree_node *t)` here, line numbers may be more accurate.