COM SCI 132 Week 9

Aidan Jan

May 29, 2024

Class Heirarchy Analysis

Suppose we have a case where class `B` contains field `m`, and `C` extends class `B`.
In this case, if we have

```
B y;
C x;
...
x.m();
y.m();
```

x.m() is valid. In this case, we make the assumption that it is a closed world, in that our source files contain all the code for the methods. In fact, any class can be the class of the object in `x`. When you do `x.m`, it searches through the class allocation in the heap for the method or field name. The compiler essentially generates a matrix containing all the class names along one axis, all the method names on the other axis, and if the method exists, a cell is filled with the memory address of the function code.
If in our scenario now, both `B` and `C` implement their own function `m`, then

```
B y;
C x;
...
x.m();
y.m();
```

would both be valid, but call different `m` methods. The compiler's method matrix would have different entries for `x.m` and `y.m`.

This type of analysis, referred to as Class Heirarchy Analysis, began in 1995, when Java was released. (The concept of inheritance was present before, just this type of analysis was new.)

Rapid Type Analysis (1995)

Suppose we have the same scenario as before. `C` extends `B`, and both classes have their own function `m`.

In rapid type analysis, it makes some structure like { Set of classes D | new D() in the program }. This represents a set (in set builder notation) of the classes present in the program, where the class has been instantiated with `new`.

If our program has one line `C x;`, then this set would only contain {`C`}, even though `C` extends `B`, but `B` is not contained in the set since it was never instantiated with `new`. By this logic, since the compiler knows that there are no `B` objects, `B.m` would never be called. This means `B.m` does not need to be dynamically loaded.

When many libraries were imported, but only certain functions in the libraries were used, this led to an average reduction in code size and compilation time by about 90%.

0-CFA

We have the grammar:

```
e ::= x
   |  new C()
   |  e1.m(e2)
s ::= x = e
```

The idea:

$$\boxed{\text{program}} \longrightarrow \boxed{\text{constraint}} \longrightarrow \boxed{\text{solution}}$$

- Use the program to generate constraints
- Solve the constraints

For our grammar, we begin with an expression e, and use it to generate (constraint) set variable $[[e]]$. The set variable ranges over sets of class names.

If we have the scenario from before, then if we have `x.m()` we would want to generate the following set: $[[x]]$ - a superset of the true content of `x`.

The following code snippets would generate the following sets:

- `new C()` $\rightarrow$ [[`new C()`]] = {`C`}
- `x = e` $\rightarrow$ [[x]] $\supseteq$ [[e]]
- e_1`.m(`e_2`)` $\rightarrow$ $[[e_1]]$, $[[e_2]]$

For every class `C` with a method `m`, [FILL]

Constraints for 0-CFA

- $\{C\} \subseteq V$, where V are set variables.
- $V \subseteq V'$
- $(C \in V) \Rightarrow (V' \subseteq V'')$

We can solve such constraints in $O(n^3)$ time.

Theorem

Class Hierarchy Analysis $\supseteq$ 0-CFA

Basically, if you're doing analysis on a program, using class hierarchy analysis may give you {A, B, C} as the set, while 0-CFA may give you {A, B}. Basically, 0-CFA would always generate a set that is a subset of or equal to the set from class hierarchy analysis.

By this logic, 0-CFA is always more powerful than class hierarchy analysis.

1-CFA

Consider the following code:

```
class P {
    (Type) s() {
        ...
    }
}

class Q extends P {
    (Type) s() {
        ...
    }
}

class B {
    (Type) m(P a) {
        a.s();
    }
}

class Main {
    psvm(String[] args) {
        x = new B(new P());
        x.m();

        x = new B(new Q());
        x.m();
    }
}
```

In this case, method `m` in class `B` can take in both `P` and `Q` objects. During compilation, this method is actually redefined twice, so separate functions are called for `P` and `Q`. (That can be a lot of copies). However, each copy can be individually optimized. This process of splitting up a method is referred to as *devirtualization*, or instead of doing *virtual* calls to a method (via class extension matching stuff), you are converting them to *direct* calls.

Alternatively, if a method is only called like once, the compiler may detect that and just copy-paste the function body to locations it is called, saving a (slow) function call statement. However, if the function contains the `this` keyword, what does it refer to? It would replace all of the `this` statements with a reference to the object calling the function. This works 100% of the time, in that it's possible to do it. The only problem is that suppose we have the case:

```
interface I {
    void m();
}

class C implements I {
    C f;
    void m() {
        this.f = this;
    }
}
```

If this is called once, and the compiler decides to copy-paste the body and replace all instances of `this`, we run into a problem. The type would now reference `I` instead of `C`, which doesn't type check! (Interfaces cannot be instantiated.) We would have to tweak the code to recognize the object as a type `C` instead.

Branch Prediction

If we have a conditional in a loop, it is likely that most times it will execute on one branch. One optimization is to compile the more common branch to be fast and the less common branch to be slow.