Lecture Notes: Object-Oriented Call Graph Construction

17-355/17-665: Program Analysis *(Spring 2017)*
Jonathan Aldrich
aldrich@cs.cmu.edu

1 Dynamic dispatch

Analyzing object-oriented programs is challenging because it is not obvious which function is called at a given call site. In order to construct a precise call graph, an analysis must determine what the type of the receiver object is at each call site. Therefore, object-oriented call graph construction algorithms must simultaneously build a call graph and compute aliasing information describing to which objects (and thereby implicitly to which types) each variable could point.

1.1 Simple approaches

The simplest approach is *class hierarchy analysis*, which uses the type of a variable, together with the class hierarchy, to determine what types of object the variable could point to. Unsurprisingly, this is very imprecise, but can be computed very efficiently in $O(n * t)$ time, because it visits n call sites and at each call site traverses a subtree of size t of the class hierarchy.

An improvement to class hierarchy analysis is *rapid type analysis*, which eliminates from the hierarchy classes that are never instantiated. The analysis iteratively builds a set of instantiated types, method names invoked, and concrete methods called. Initially, it assumes that `main` is the only concrete method that is called, and that no objects are instantiated. It then analyzes concrete methods known to be called one by one. When a method name is invoked, it is added to the list, and all concrete methods with that name defined within (or inherited by) types known to be instantiated are added to the called list. When an object is instantiated, its type is added to the list of instantiated types, and all its concrete methods that have a method name that is invoked are added to the called list. This proceeds iteratively until a fixed point is reached, at which point the analysis knows all of the object types that may actually be created at run time.

Rapid type analysis can be considerably more precise than class hierarchy analysis in programs that use libraries that define many types, only a few of which are used by the program. It remains extremely efficient, because it only needs to traverse the program once (in $O(n)$ time) and then build the call graph by visiting each of n call sites and considering a subtree of size t of the class hierarchy, for a total of $O(n * t)$ time.

1.2 0-CFA Style Object-Oriented Call Graph Construction

Object-oriented call graphs can also be constructed using a pointer analysis such as Andersen's algorithm, either context-insensitive or context-sensitive. The context-sensitive versions are called k-CFA by analogy with control-flow analysis for functional programs (to be discussed in a forthcoming lecture). The context-insensitive version is called 0-CFA for the same reason. Essentially,

the analysis proceeds as in Andersen's algorithm, but the call graph is built up incrementally as the analysis discovers the types of the objects to which each variable in the program can point.

Even 0-CFA analysis can be considerably more precise than Rapid Type Analysis. For example, in the program below, RTA would assume that any implementation of foo() could be invoked at any program location, but O-CFA can distinguish the two call sites:

```
class A { A foo(A x) { return x; } }
class B extends A { A foo(A x) { return new D(); } }
class D extends A { A foo(A x) { return new A(); } }
class C extends A { A foo(A x) { return this; } }

// in main()
A x = new A();
while (...)
    x = x.foo(new B()); // may call A.foo, B.foo, or D.foo
A y = new C();
y.foo(x);                  // only calls C.foo
```

Acknowledgements

I thank Claire Le Goues for greatly appreciated extensions and refinements to these notes.