Method Based Capability Control

20th July 2018

In imperative languages, reasoning on side-effects can be very challenging, since any piece of code can potentially do anything. In object oriented systems where dynamic dispatch is pervasive, we do not even have access to such code. For example, executing the innocent looking method can format your hard drive.

Object capabilities allow delegating reasoning on side effects, by reasoning on aliasing: only special unforgeable objects can do special actions. Then, if reasoning over aliasing proves that the reachable-object-graph of does not contain a (or stronger) capability, then we can be certain calling will not format the hard drive.

In this paper we explore alternative approach, where methods can be called only when sufficient permissions are available. For simplicity we will use class and method names as permission labels, but conceptually any kind of label would do. By default a methods permissions require only the labels of methods they directly invoke. However method declarations can chose to declare more permissions, in-order to restrict their usage and aid maintainability. For example, if our standard library provides a class:

Then every function in the system could read any kind of file. We can use permissions to change this:

Now, only when the permission is in scope can be (directly) called; for example this is correct:

acts as a filter, and reads only files presents in the 'Documents' folder. Of course we can call if we have the , and, permissions. However, merely having the permission does not give us the permission, and so we cannot directly call it:

In this way, by reasoning on the code of , we can understand how the power of is tamed, in particular, we can guarantee that code that only has the permission can only read files in the documents folder. This is similar to object-capabilities where one would provide a object with a (private) field, and a would be an instance method.

For convenience, we allow classes to define a set of '*implied*' permissions: in this way the class name is just shorthand for those other permissions. For example, if Directory was declared as:

then, while declaring before, we could have written instead of

The system as presented up to now is completely static and does not require nor take advantage of objects. Every method requires an exact set of permissions and thus can do a specific set of actions. Using generics, subtyping, and objects we can write permission generic code, in true object-capability style:

Notice that unlike a traditional object-capability system, we need not restrict the use of the above constructors, since we can independently restrict methods; however we can still do so to properly enforce the object-capability pattern. With these classes defined, one can now write a parametric method :

Notice that stands for a list of permissions, not a single one, however these annotations could be inferred. The call is valid since has permission , which is more than sufficient to call . Following the presented pattern, the programmer has control on how much static information they require and how much they are ready to rely on dynamic (aliasing based) control. For example, we could declare:

And now, our capability object can do less (but not more) then our capability , but from a static perspective there is no difference.

An important benefit of our approach is that it allows safety and control even in the case of static methods performing primitive operations. This is very useful with making native calls; with the simple restriction that native calls need to correspond to restricted static method calls, we believe our system allows encoding safe object capabilities as a user library, instead of requiring them to be integrated in the standard library.