Method Based Capability Control

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 know what code could be run. For example, executing the innocent looking method `foo` can format your hard drive.

```
void foo(Point myPoint) { myPoint.getX(); }
```

Object capabilities delegate reasoning on side effects to reasoning on aliasing. Since only special unforgeable objects can do special actions, if reasoning over aliasing proves that the reachable-object-graph of `myPoint` does not contain a `FileSystem` capability, we can be certain calling `foo(myPoint)` will not format the hard drive.

Here we explore an alternative approach, where methods can only be called when sufficient permissions are available. We will use class and method names as permission labels. A method declaration is annotated with it's permissions set Ps. The method can only call itself, the methods whose names are in it's permissions set, and any method whose permission set is a subset of Ps. It follows that any method requiring no permissions can always be called. For example, if our standard library provides a class:

```
class System {
  static String readFile(String fileName){/*magic implementation*/}
}
```

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

```
class System {
  @Permissions(System.readFile) // permission label!
  static String readFile(String fileName){/*magic implementation*/}
}
```

A method can declare *more* permissions than needed to call all the methods used in its body. This is the way one introduce restrictions: if no method ever declared any permission, no permission would ever be required.

Now only within a method whose permissions include `System.readFile` can `System.readFile` be *directly* called; for example this is correct:

```
class Documents {
  @Permissions(System.readFile, Directory.new, Directory.contains)
  static String readFile(String fileName) {
    if (!new Directory("~/Documents").contains(fileName)){throw /*error*/;}
    return System.readFile(fileName);
  }
}
```

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

```
@Permissions(Documents.readFile)
static void main() {
  String doc1 = Documents.readFile("~/Documents/hi.txt");
  //String doc2 = System.readFile("~/Documents/hi.txt");//Invalid!
}
```

In this way, by reasoning on the code of `Documents.readFile`, we can understand how the power of `System` is tamed, in particular, we can guarantee that code which only has the `Documents.readFile` permission can only read files in the documents folder.

This is a static and method level re-interpretation of the common object capability technique using the delegation pattern: where a new object wraps the capability object and performs restrictions and checks while delegating the method behaviour.

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:

```
@Implies(Directory.new, Directory.contains)
class Directory {...}
```

then, while declaring `Documents.readFile` before, we could have written ‘`Directory`’ instead of ‘`Directory.new, Directory.contains`’.

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 dynamic-dispatch we can write permission generic code, in true object-capability style:

```
interface IFiles[A] {
  @Permissions(A)
  String readFile(String fileName);
}
class Files implements IFiles[System.readFile] {
  @Permissions(Files.new)
  Files(){}

  @Permissions(System.readFile)
  System.readFile String readFile(String fileName){return System.readFile(fileName);}
}
class DocFiles implements IFiles[Documents.readFile] {
  @Permissions(DocFiles.new)
  DocFiles(){}

  @Permissions(Documents.readFile)
  String readFile(String fileName){return Documents.readFile(fileName);}
}
class MockFiles implements IFiles[] {
  MockFiles(){}//if @Permissions is omitted, of course it means the empty set

  String readFile(String fileName) { return "";}
}
```

With these classes defined, one can now write a parametric method `foo`:

```
@Permissions(A)
[A] String foo(IFiles[A] cap) {
  // Internally, cap can be used without static knowledge of what it can do
  return cap.readFile("foo.txt");
}
```

Notice that **`A`** stands for a list of permissions, not a single one. We believe that these annotations could often be inferred. The call `cap.readFile("foo.txt")` is valid since `foo` has permission **`A`**, which is more than sufficient to call **`IFiles[A]`**`.readFile`. For example, the call `foo(new DocFiles())` would be inferred to `foo[Documents.readFile](new DocFiles())`, and then the body of `foo` would have the **`Documents`**`.readFile` permission; this is sufficient for it to call `cap.readFile("foo.txt")`.

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:

```
class FilesStart extends Files {
  @Permissions(Files.new)
  FilesStart(){super();}

  @Permissions(System.readFile)
  String readFile(String fileName){return super.readFile(fileName).substring(0,10);}
}
```

Notice how both **`Files`**`.readFile` and **`FilesStart`**`.readFile` can directly call **`System`**`.readFile`. Thus, thanks to dynamic dispatch, from a static reasoning perspective, a call to **`Files`**`.readFile` is as powerfull as a call to **`System`**`.readFile`. However, since **`Files`**`.new` is restricted and constructors need to call super-constructors, we may be able to refine our understanding of a specific **`Files`**`.readFile` call, by reasoning over aliasing. On the other hand, **`DocFiles`**`.readFile` and all overrides of it are bound to only call **`Documents`**`.readFile`.

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 in conjunction with native calls; with the simple restriction that native functions correspond to static methods which have their names within their permission set. With such restriction, we believe our system allows encoding safe object capabilities as a user library, instead of requiring them to be integrated in the standard library.