What's in an ion?

Simon Dobson

February 1999

This paper describes the the first step towards implementing ionic classes.

1 Motivation

The idea for an ion arose from a simple question: why is sub-classing regarded as a compile-time operation? The traditional answer to this question is that, by evaluating the sub-class operator early, the compiler can optimise the generated code. A good example is C++'s use of virtual function tables, where references to named object members are replaced by faster numeric indexing. Although this improves performance it makes analysis of compiled code all but impossible.

Language such as Java adopt a solution which might be regarded as the worst of all worlds. Java sub-classing is a compile-time operation — although it is possible to construct new classes at run-time, it is *only* possible by generating the appropriate compiled bytecodes and not through the Java language itself. However, dynamic linking means that the Java compiler must use name look-up for resolving member accesses.

Instead we may look at sub-classing as a run-time operation (which effectively implies that classes must be denotable values). This then allows us to construct *ions*: abstractions over a super-class (with suitable polymorphic constraints) which allows the same sub-class code to be applied uniformly to a range of super-classes.

2 Ion type theory

We claim that a system with ions is still type-safe, despite the under-commitment to ion base classes. Proving this claim requires that we express the idea of ions in a suitable object calculus. We choose to implement ions using the language O-2 of Abadi and Cardelli[1]. The advantage of this is that O-2 is a direct realisation of the second-order object calculus $S_{\varsigma\forall}$ so an implementation of ions in O-2 provides an immediate hook into their formalisation. A further advantage is that we have a readily-available implementation of O-2 in Vanilla.

The first point is that ions are definable types within the type system of O-2, and need not be treated as primitive. This is a consequence of O-2's treatment of classes as objects under a particular encoding (making the class available at run-time) and its use of bounded polymorphic functions (to allow type constraints to be expressed). Ions are *not* directly definable in C++ or Java.

To illustrate the implementation, let us begin by defining a simple "point" class:

```
 Point === Object(X) [ x :  Int, y :  Int, mv :  Int
-> Int -> X ]
pointClass :  Class(Point) === subclass of root :
Root with (self :  X <:  Point) x = 0, y = 0, mv =
fun( dx :  Int ) fun( dy :  int ) (self.x := self.x +
dx).y := self.y + dy end end
```

Two parts of this code require explanation, neither point being critical to what follows. Firstly, assignment in O-2 is functional, so `self.x := self.x + dy` gives rise to a new object which is a copy of `self` with the `x` member incremented. Secondly the type variable `X` is a *covariant self type* which always refers to the type of the receiving object. This means that `mv` can never return an object whose type is wider than that of the receiver. This is somewhat different from the case in Java, where the similar construct (mentioning a class' name in its method signatures) would allow a method to return an object as wide as the original defining type.

We now define an ion which adds a `translate()` method to any class which has a `mv()` method. We begin by defining the necessary types for an object with `mv()` and the type resulting from our extending the `Point` class:

```
HasMove === Object(X) [ mv :  Int -> Int -> X ]
TranslatingPoint === Object(X) [ x :  Int, y :
Int, mv :  Int -> Int -> X, translate :  Int -> X ]
```

The ion may then be coded as:

```
TranslateIon === All(A <:  HasMove) All(B <:  A)
Class(A) -> Class(B)
translateIon :  TranslateIon === fun(A <:  HasMove)
fun(B <:  A) fun( aclass :  Class(A) ) subclass of
aclass :  Class(A) with (self :  X <:  B) translate =
fun( d :  Int ) self.mv(d)(d) end end end end end
```

What is this doing? The `TranslateIon` type states that the ion is polymorphic in two types: a type `A` representing any type with `mv()` and a further type `B` extending this type. The ion itself is a function from a class of `A` to

a class of `B`. In other words the ion takes a suitably-constrained base class and returns an extension of it.

The encoding itself is fairly straightforward, generating a sub-class (at run-time) of the parameter class:

```
translatingPointClass :  Class(TranslatingPoint) ===
translateIon(Point)(TranslatingPoint)(pointClass)
tp :  TranslatingPoint === new translatingPointClass
tp.translate(10)
```

To show that ionic extension works with different base types, we can apply the same ion to another class, a sub-class of `Point` adding a colour string:

```
CPoint === Object(X) [ c :  String, x :  Int, y :
Int, mv :  Int -> Int -> X ]
cPointClass :  Class(CPoint) === subclass of
pointClass :  Class(Point) with (self :  X <:
CPoint) c = "black" end
TranslatingColouredPoint === Object(X) [ c :  String,
x :  Int, y :  Int, mv :  Int -> Int -> X, translate
:  Int -> X ]
translatingColouredPointClass :
Class(TranslatingColouredPoint) ===
translateIon(CPoint)(TranslatingColouredPoint)(cPointClass)
tcp :  TranslatingColouredPoint === new
translatingColouredPointClass tcp.translate(10).c
```

One way of looking at this is as follows: normally to add functionality uniformly to a set of classes it is necessary to re-code one of their common ancestor classes (assuming they have one). Ions invert this, allowing us to apply the *same* functionality below *different* points of the inheritance hierarchy.

3 Applications and next steps

There are several potential applications for ions, including:

- *Uniform extension* — adding the same functionality uniformly to a set of classes without requiring access to the source of their base classes and without their necessarily sharing a common ancestor.
- *Base class re-assignment* — this causes horrendous foundational problems. An ion may be used to achieve a similar effect by applying the (abstracted) sub-class to a functionally different base class.

- *Security in mobile environments* — we may encapsulate a class' access to its environment using an ion. The type constraint determines the operations which can access the environment; applying the ion to a base class and then instantiating fixes the objects' possible interactions with the environment without the need for run-time security checks.,

The next step is to formulate ions into a standard framework such as our "decaffeinated" Java. This will require a direct axiomatisation of the ion idea. We may then explore how ions interact syntactically with the commonly-available object-oriented language features.

4 References

References

[1] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer Verlag, 1996.