Essentials of Scala

Martin Odersky

Ecole Polytechnique Fédérale de Lausanne

Summary

Languages like Scala fuse object-oriented programming with concepts from module systems. Like an ML module, an object can have types as members. This poses new challenges for the type theoretic foundation of these languages. Instead of a stratified solution à la ML, we are looking for a system that does not distinguish between core language and module system.

In this talk I'll present a new type-system that can model Scala and languages like it. The properties we are interested in are Scala's path-dependent types and abstract type members, as well as its mixture of nominal and structural typing through the use of refinement types. Compared to previous approaches, we make no attempt to model inheritance or mixin composition. Indeed we will argue that such concepts are better expressed in a different setting.

The calculus does not precisely describe what's currently in Scala. It is more normative than descriptive. The main point of deviation concerns the difference between Scala's compound type formation using **with** and classical type intersection, as it is modelled in the calculus. Scala, and the previous calculi attempting to model it conflates the concepts of compound types (which inherit the members of several parent types) and mixin composition (which build classes from other classes and traits). At first glance, this offers an economy of concepts. However, it is problematic because mixin composition and intersection types have quite different properties. In the case of several inherited members with the same name, mixin composition has to pick one which overrides the others. It uses for that the concept of a *linearization* of a trait hierarchy. Typically, given two independent traits with a common method, the mixin composition of those traits would favor the method definition in the second trait, whereas the member in the first trait would be accessible with a super-call. All this makes sense from an implementation standpoint. From a typing standpoint it is more awkward, because it breaks commutativity and with it several monotonicity properties.

In the proposed calculus, we replace Scala's compound types by classical intersection types, which are commutative. We also complement this by classical union types. Intersections and unions form a lattice wrt subtyping. This addresses another problematic feature of Scala's current type system, where least upper bounds and greatest lower bounds do not always exist.