Future-Proofing Collections: From Mutable to Persistent to Parallel

Martin Odersky

Programming Methods Group (LAMP)
School of Computer and Communication Sciences (IC)
École Polytechnique Fédérale de Lausanne (EPFL)
Lausanne, Switzerland
`martin.odersky@epfl.ch`

Abstract. Multicore processors are on every desk now. How are we going to make use of the extra power they provide? Some think that actors, or transactional memory, or some other new concurrency construct will save the day by making concurrent programming easier and safer. Even though these are welcome, I am skeptical about their ultimate success. Concurrency is fundamentally hard and no dressing up will be able to hide that fact completely.

A safer and for the programmer much simpler alternative is to treat parallel execution as essentially an optimization. A promising application area are collections. Programming by transforming and aggregating collections is simple, powerful, and can be optimized by executing bulk operations in parallel. To be able to do this in practice, any side effects of parallel operations need to be carefully controlled. This means that immutable, persistent collections are more suitable than mutable ones.

In this talk I will describe the new Scala collections framework, and show how it allows a seamless migration from traditional mutable collections to persistent collections, and from there to parallel collections. I show how the same vocabulary of methods can be used for either type of collection, and how one can have parallel as well as sequential views on the same underlying collection.

The design of this framework is guided by the "uniform return type principle": every collection transformer should return the same kind of collection it applies to. Simple as this sounds, it is surprisingly hard to achieve in a statically typed language with a rich type hierarchy (in fact, I know of no other collections framework that achieved it). In the talk I will explain the type-systematic constructions required by the principle. I will also present some thoughts on how we might develop type-explanation capabilities of compilers to effectively support these techniques in a user-friendly way.