Effects as Capabilities

Fengyun Liu, Nicolas Stucki, Sandro Stucki, Nada Amin and Martin Odersky
LAMP, EPFL: {first.last}@epfl.ch

It seems quite natural that one should track effects by means of a static typing discipline, similarly to what is done for arguments and results of functions. After all, to understand a function's contract and how it can be composed, knowing its effects is just as important as knowing the types of its arguments and result. Yet after decades of research [3, 4, 9, 10, 11, 2, 8, 1], why are effect systems not as mainstream as type systems?

The static effect discipline with the most widespread use is no doubt Java's system of checked exceptions. Ominously, they are now widely regarded as a mistake [12]. One frequent criticism is about the notational burden they impose. Throws clauses have to be laboriously threaded through all call chains. All too often, programmers make the burden go away by catching and ignoring all exceptions that they think cannot occur in practice. In effect, this disables both static and dynamic checking, so the end result is less safe than if one started with unchecked exceptions only. Another common problem of Java's exceptions is lack of polymorphism: Often we would like to express that a function throws the same exceptions as the (statically unknown) functions it invokes. Effect polymorphism can be expressed in Java only at the cost of very heavy notation, so it is usually avoided. Java's system of checked exceptions may be an extreme example, but it illustrates the general pitfalls of checking effects by shifting the burden of tracking effects to the programmer.

We are investigating a new approach to effect checking, that flips the requirements around. The central idea is that instead of talking about effects we talk about *capabilities*. For instance, instead of saying a function "throws an `IOException`" we say that the function "needs the capability to throw an `IOException`". Capabilities are modeled as values of some capability type. For instance, the aforementioned capability could be modeled as a value of type `CanThrow[IOException]`. A function that might throw an `IOException` needs to have access to an instance of this type. Typically it takes an argument of the type as a parameter.

It turns out that that the treatment of effects as capabilities gives a simple and natural way to express "effect polymorphism" – the ability to write a function once, and to have it interact with arguments that can have arbitrary effects. Since capabilities are just function parameters, existing language support for polymorphism, such as type abstraction and subtyping, is readily applicable to them. But there are two areas where work is needed to make capabilities as effects sound and practical.

First, when implemented naively, capabilities as parameters are even more verbose than effect declarations such as throws clauses. Not only do they have to be declared, but they also have to be propagated as additional arguments at each call site. We propose to make use of the concept of *implicit parameters* [5, 6, 7] to cut down on the boilerplate. Implicit parameters make call-site annotations unnecessary, but they still have to be declared just like normal parameters. To avoid repetition, we propose to investigate a way of abstracting implicit parameters into *implicit function types*. With implicits, the approach provides the common case of propagation for free, and an easy migration path from impure to pure.

Second, there is one fundamental difference between the usual notions of capabilities and effects: capabilities can be captured in closures. This means that a capability present at closure construction time can be preserved and accessed when the closure is applied. Effects on the other hand, are temporal: it generally does make a difference whether an effect occurs when a closure is constructed or when it is used. We propose to address this discrepancy by introducing a "pure function" type, instances of which are not allowed to close over effect capabilities.

In this talk, we report on work in progress, exploring the idea of effects as capabilities in detail. We have worked on minimal formalizations for implicit parameters and pure functions and studied encodings of higher-level language constructs into these calculi. Based on the theoretical modelization we are developing a specification for adding effects to Scala.

References

[1] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[2] Andrzej Filinski. Monads in action. In *POPL*, 2010.

[3] David K Gifford and John M Lucassen. Integrating functional and imperative programming. In *POPL*, 1986.

[4] John M Lucassen and David K Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

[5] Martin Odersky. Poor man's typeclasses. Presentation to IFIP WG 2.8, 2006.

[6] Bruno CdS Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

[7] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: A new foundation for generic programming. In *PLDI*, 2012.

[8] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *ECOOP*. 2012.

[9] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming (JFP)*, 2(03):245–271, 1992.

[10] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.

[11] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)*, 4(1):1–32, 2003.

[12] Thomas Whitmore. Checked exceptions, Java 's biggest mistake. Literal Java Blog, 2015.