Representing Monads with Capabilities

Jonathan Immanuel Brachthäuser
EPFL, Switzerland

Aleksander Boruch-Gruszecki
EPFL, Switzerland

Martin Odersky
EPFL, Switzerland

Abstract

Programming with monads can be advantageous even in imperative languages with builtin support for side effects. However, in these languages composing monadic programs is different from composing side effecting imperative programs. This does not need to be the case, as already noticed by Filinski [1994]. We revive the well-known technique of *monadic reflection* in the context of modern programming languages with support for fibers, generators, or coroutines. In particular, we show how (layered) monadic reflection can be implemented in a stack safe manner and how effect safety can conveniently be approximated by capability passing.

Comments This is a talk proposal, accompanying our talk at the Workshop on Higher-order Programming with Effects (HOPE 2021).

Prelude

In his seminal paper on functional programming with monads, Wadler [14] asks

> Shall I be pure or impure?

A question that divided programming languages (and communities) back then and still does to the current day.

In his foundational work, Filinski [6] notices that

> It is somewhat remarkable that monads have had no comparable impact on "impure" functional programming.

In an attempt to close the gap between imperative and functional programming, Filinski presents the technique of *monadic reflection*. If a language has support for *composable contuations* (such as offered by the control operators shift/reset [4]), then it can represent arbitrary control effects – a fact that earned the CPS monad the title "Mother of All Monads". Given composable continuations, Filinski shows that it is possible to define the two operators reflect (*i.e.*, $\mu(\ \cdot\ )$) and reify (*i.e.*, $[\ \cdot\ ]$):

$$\frac{\Gamma\ \vdash\ \ E\ :\ T\ \alpha}{\Gamma\ \vdash\ \ \mu(E)\ :\ \alpha}\ [\textsc{reflect}] \qquad\qquad \frac{\Gamma\ \vdash\ \ E\ :\ \alpha}{\Gamma\ \vdash\ \ [\ E\ ]\ :\ T\ \alpha}\ [\textsc{reify}]$$

For some monadic type constructor T, reflecting allows *embedding* a monadic computation of type $T\ \alpha$ into the current computation. Dually, reifying allows programmers to make the monadic structure of a computation explicit.

Monadic Reflection, Today

Monadic reflection is potentially more relevant than ever. Firstly, while monads do allow programming with and reasoning about effects in pure programming languages, they did not resolve the divide between functional and imperative programming. Quite the contrary, as can be observed in strongly typed (impure) languages like Scala: even though Scala natively supports effects, programmers fall back to writing in monadic style in order to reason about effectful programs as *referentially transparent* values. Notably, monadic style prevents programmers to directly use built-in mechanimns to structure the control flow, such as imperative sequencing (*i.e.*, `;`), loops, exceptions, and existing higher-order functions (*e.g.*, `map` or `foreach`). Secondly, while to the current day only a few languages support delimited control operators like `shift/reset`, as required by Filinski, more and more languages are equipped with alternative features to structure non-local control flow, such as generators [12], async/await [1], coroutines [9], or fibers [5, 13]. As shown by James and Sabry [8], many of these constructs can be used to encode delimited control operators. Maybe unsurprisingly, languages like JavaScript, Python, C#, and JVM languages (like Java, Scala, etc.) thus already provide the necessary means to implement monadic reflection.

Shall I be pure or impure? Why not both?

While being an important theoretical result, we believe the practical implications of the work by Filinski [6] are severly underrecognized. Monadic reflection equips programmers with a tool to seamlessly switch between imperative programming (using all existing mechanisms to structure control flow) and monadic programming (reifying computation to reason with referential transparency). In this talk, we follow Filinski and take the position that instead of improving the support for programming with monadic values (such as `do`-notion), imperative programming constructs should be used to construct effectful programs, only reifying the monad when necessary (for instance for non-standard composition of effectful programs). To demonstrate the practicality of monadic reflection, we combine the work of Filinski and James and Sabry and show how fibers [13] can be used to implement (layered) monadic reflection [7] as a library in the Scala 3 language. Our implementation is a proof of concept, suggesting that monadic reflect is readily available in languages with support for fibers, generators, or async/await. The library can immediately be used in the context of existing functional programming ecosystems such as *scala cats*[1], *scalaz*[2], or *ZIO*[3], replacing convoluted monadic composition via for-comprehensions with direct-style code.

Capability-Based Monadic Reflection

Filinski [7] already remarks that

> a type system for actually enforcing the effect-restrictions statically would be a big help in constructing large programs. [. . .] it should also include support for some notion of effect-polymorphism.

With its support for contextual abstractions[4], Scala 3 provides an approximate, yet convenient alternative to an effect system [10, 2]. For example, the context function type `Eff ?=> Res` denotes values of type `Res` that can only be used in contexts where an implicit `Eff` is in scope. We often refer to values of type `Eff` as *capabilities* since they can be seen as the constructive proof that a holder is entitled to perform actions on `Eff`. It has been shown

[1] https://typelevel.org/cats/
[2] https://scalaz.github.io/7/
[3] https://zio.dev/
[4] Contextual abstractions were formerly known as *implicits*.

```
trait Reflect[M[_]] { def reflect[A](ma: M[A]): A }
def reify[M[_]: Monadic, A](
  prog: Reflect[M] ?=> A
): M[A]
```

```
trait Monadic[M[_]] {
  def pure[A](a: A): M[A]
  def sequence[X, R](mx: M[X])(
    f: X => Either[M[X], M[R]]): M[R]
}
```

Figure 1 Interface for capability-based layered monadic reflection in Scala 3. To reify a computation for a type constructor `M`, an implementation of the type class `Monadic[M]` is necessary. The method `seq` is a stack-safe variant of traditional monadic composition. Reifying a monad `M` introduces a capability `Reflect[M]` that can be used to reflect computation of type `M[A]`.

that basing effect systems on capabilities opens up interesting and light-weight forms of effect polymorphism [11, 3].

In our implementation of monadic reflection (summarized in Figure ??), we make use of contextual function types in the following way. Reifying a monadic computation of the type constructor `M[_]` introduces an implicit capability `Reflect[M]` in the scope of the reified computation. Within that scope, the capability can be used to reflect values of type `M[A]`. The following example uses monadic reflection to express errors in the `Option` monad.

```
def abort[A]()(using r: Reflect[Option]): A =
  r.reflect(None)

def safeDiv(x: Int, y: Int)(using Reflect[Option]): Int =
  if (y == 0) abort() else x / y
```

```
reify[Option, Int] {
  safeDiv(4, 0) + safeDiv(5, 0)
}
```

Calling the function `abort` requires an implicit instance of `Reflect[Option]` to be available at the callsite. On the other side, as becomes visible in the type signature of `reify` (Figure ??), calling `reify[M, A]` brings such an implicit instance of type `Reflect[M]` into scope.

Representing Layered Monads via Multiple Prompts

The implementation of fibers in Project Loom [13] allows labeling individual fibers with what could be seen as *prompts*. When yielding, we can choose which prompt to suspend to. This native support for multi-prompt delmited control immediately gives rise to an efficient implementation of *layered* monadic reflection [7]. Calls to `reify` can be nested, introducing seperate capabilities. Calling `reflect` on a capability will immediately transfer control to the correct `reify` call. This is illustrated in the combined use of `Option` and `Future`:

```
def read(f: String)(using r: Reflect[Future]): Source =
  r.reflect(Future { Source.fromFile(f) })

reify[Option, Int] {
  Await.result(reify[Future, Int] {
    read("test.txt").size + safeDiv(5, 0)
  }, 100 milliseconds)
}
```

References

1 G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause'n'play: Formalizing asynchronous C#. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 233–257, Berlin, Heidelberg, 2012. Springer.

2 J. I. Brachthäuser and P. Schuster. Effekt: Extensible algebraic effects in Scala (short paper). In *Proceedings of the International Symposium on Scala*, New York, NY, USA, 2017. ACM. doi: 10.1145/3136000.3136007.

3 J. I. Brachthäuser, P. Schuster, and K. Ostermann. Effects as capabilities: Effect handlers and lightweight effect polymorphism. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020. doi: 10.1145/3428194.

4 O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the Conference on LISP and Functional Programming*, pages 151–160, New York, NY, USA, 1990. ACM.

5 S. Dolan, S. Eliopoulos, D. Hillerström, A. Madhavapeddy, K. Sivaramakrishnan, and L. White. Effectively tackling the awkward squad. In *ML Workshop*, 2017.

6 A. Filinski. Representing monads. In *Proceedings of the Symposium on Principles of Programming Languages*, page 446–457, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.178047. URL `https://doi.org/10.1145/174675.178047`.

7 A. Filinski. Representing layered monads. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 175–188, New York, NY, USA, 1999. ACM.

8 R. P. James and A. Sabry. Yield: Mainstream delimited continuations. In *First International Workshop on the Theory and Practice of Delimited Continuations (TPDC 2011)*, volume 95, page 96, 2011.

9 A. L. D. Moura and R. Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, Feb. 2009. ISSN 0164-0925.

10 M. Odersky, O. Blanvillain, F. Liu, A. Biboudis, H. Miller, and S. Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM Program. Lang.*, 2 (POPL):42:1–42:29, Dec. 2017. ISSN 2475-1421.

11 L. Osvald, G. Essertel, X. Wu, L. I. G. Alayón, and T. Rompf. Gentrification gone too far? affordable 2nd-class values for fun and (co-) effect. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 234–251, New York, NY, USA, 2016. ACM.

12 J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 217–232, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1.

13 R. Pressler. Loom Project: Fibers and Continuations for the Java Virtual Machine. OpenJDK Project, HotSpot Group, September 2017. URL `http://mail.openjdk.java.net/pipermail/discuss/2017-September/004390.html`.

14 P. Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, pages 24–52. Springer, 1995.