Effekt: Extensible Algebraic Effects in Scala
(Short Paper)

Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Philipp Schuster
University of Tübingen, Germany

Abstract
Algebraic effects are an interesting way to structure effectful programs and offer new modularity properties. We present the Scala library Effekt, which is implemented in terms of a monad for multi-prompt delimited continuations and centered around capability passing. This makes the newly proposed feature of implicit function types a perfect fit for the syntax of our library. Basing the library design on capability passing and a polymorphic embedding of effect handlers furthermore opens up interesting dimensions of extensibility. Preliminary benchmarks comparing Effekt with an established library suggest significant speedups.

CCS Concepts • Software and its engineering → Control structures;

Keywords algebraic effects, effect handlers, continuations, implicits, capabilities, shallow embedding

ACM Reference Format:

1 Introduction
Consider the following piece of code that uses two effect operations flip for nondeterministic coin flipping and raise for exception raising:

```scala
val prog : Boolean using Amb and Exc =
  if (x ≦ 0) flip() else raise("too big")
```

It expresses its use of algebraic effects in the type of `prog` by mentioning the corresponding effect signatures `Amb` and `Exc`. The program leaves open the concrete semantics of the effect operations.

We can, for instance, use the effect handler `AmbList` to handle the `Amb` effect and the effect handler `Maybe` to handle the `Exc` effect. The `AmbList` handler gathers all results of a nondeterministic computation into a list and the `Maybe` handler returns `None` if the program raises an exception.

```scala
val res1 : C[Option[List[Boolean]]] = Maybe [AmbList [ prog ]]
val res2 : C[List[Option[Boolean]]] = AmbList [ Maybe [ prog ]]
```

We can see in the type of results that the order of handling matters, a distinguishing feature of algebraic effects with handlers. Running the two programs (for `x ≦ 0`) will result in `Some(List(true,false))` and `List(Some(true),Some(false))`.

What looks like code written in an extension of Scala enhanced with algebraic effects and handlers, is actually Dotty using our library Effekt\(^1\), which we present in this paper. Effekt is based on a monad for multi-prompt delimited continuations [Dyvbig et al. 2007] and a shallow embedding [Hofer et al. 2008] of effect handlers.

In the development of the library, we always strived for simplicity from the perspective of an effect user and tried to minimize type annotations and advanced type level programming. We made design decisions in favor of passing down information which integrates well with Dotty’s support for implicit function types and provides evidence of the usefulness of this feature.

The combination of recently proposed Scala features (implicit function types, second class values) and well established Scala features (path dependent types and singleton types) provides a good foundation for a library for algebraic effects with handlers. We believe that the properties of its design can lead to a wider adoption of algebraic effects and handlers in Scala and object oriented programming in general.

In particular, this paper makes the following contributions:

- **section 2** introduces the usage of our library, explains how it uses implicit function types and elaborates on the above examples.
- **section 3** relates algebraic effects and handlers to the expression problem, explores several dimensions of extensibility and shows how Effekt supports them.
- **section 4** compares our design decision in favor of capability passing to other solutions from related work. It further sketches how second class values can help to improve the safety of our library. Finally, it reports preliminary performance results.

We hope that this short paper can draw the attention of the Scala community to effect implementation techniques other than those based on freer monads and that it motivates future research for better language support of algebraic effects as a library.

---

\(^1\) [http://b-studios.de/scala-effekt/](http://b-studios.de/scala-effekt/)

2 Programming with Algebraic Effects and Handlers in Effekt

To introduce our library “Effekt”, we elaborate on the introductory example and show the definition and usage of two simple effects which are standard in literature: exceptions and ambiguity. We choose to present all code examples in Dotty, a variant of the Scala language, due to its support for implicit function types and inferred eta-expansion. We use this feature only for syntactic convenience, it does not affect the operational semantics which is independent of the Dotty implementation.

Regarding the concrete syntax, the Koka language [Leijen 2014, 2017b] served as a source of inspiration for our library design. To allow easy comparison, the names of effects and handlers in this example are close to Leijen’s [2017b].

2.1 Exceptions

Let’s consider the following program which makes use of a simple exception effect:

```scala
def div(x: Int, y: Int): Int using Exc =
  if (y == 0) raise("y is zero") else pure(x / y)
```

An effectful program that uses effect E to compute a value of type A has type A using E. Here, using is a binary type alias written infix. It is defined in Figure 1 which shows the complete public interface of Effekt. All effectful programs use an underlying monad C (read “Control”), provided by our library. Hence in the else-block of the example the result needs to be lifted into C using the function pure.

Algebraic effects encourage modularity by separating the declaration of the effect signature from its semantics. Effect signatures, such as Exc are declared as a trait inheriting from Eff. Effect operations provided by the signature have type Op[A] where A is the type of the value returned by the effect operation.

```scala
trait Exc extends Eff {
  def raise[A](msg: String): Op[A]
}
```

This is not a coincidence: We view handling an algebraic effect as folding a handler over a tree of effect operations. The domain of our interpretation is always C[Res] for different result types Res depending on the handler.

For convenient use, we also define a wrapper function for every effect operation:

```scala
def handle[h: Handler[_, _]](f: h.R using h.type): C[h.Res] =
def run[A](c: C[A]): A
```

Regarding the concrete syntax, the Koka language [Leijen 2014, 2017b] served as a source of inspiration for our library design. To allow easy comparison, the names of effects and handlers in this example are close to Leijen’s [2017b].

```scala
trait Eff { type Op[A] }

trait Cap[A] { val handler: Handler[_, _] with E }

trait Handler[R0, Res0] extends Eff {
  type R = R0; type Res = Res0
  def unit: R ⇒ Res
}
```

Similar to the wrapper functions for effect operations we also wrap the constructor for the handler:

```scala
  handle(new Maybe[R]())(f)
```

We can use the Maybe handler and run an example program

```scala
run { Maybe { div(4, 0) } } to yield None.
```

2.2 Ambiguity

Our interpretation of the exception effect discards the continuation of the program when it encounters a raise and immediately returns None. The implementation of handlers for other effects requires access to the continuation in order to call it once or even multiple times.

```scala
trait Amb extends Eff {
  def flip(): Op[Boolean]
}

trait AmbList[R] extends Amb with Handler[R, List[R]]
```
The above code shows the implementation of an ambiguity effect with a coin flipping operation. To compute a list of all possible outcomes, the handler AmbList calls the continuation on true and on false and concatenates the results. We also use implicit function types in the definition of the type alias CPS[A, Res], which represents a cps transformed term of type A with answer type Res. To allow concise handler implementations, the continuation of type A ⇒ C[Res] is marked as implicit. This way, it can implicitly be passed as second argument k to the library function resume.

In the introductory example, we have seen how to use both effect operations flip and raise in the same program, combining Exc and Amb.

2.3 Implicit Function Types for Syntax

Implicit function types reduce the syntactic noise when passing down contextual information [Ódersky et al. 2017]. Currently, Scala allows to mark arguments of lambdas as implicit and thus make the argument implicitly available within the body. However, this information cannot be expressed in the type of the lambda, resulting in two problems: Firstly, it is not possible to abstract over repeating patterns of implicit arguments, as we do with the type alias using. Secondly at the binding occurrence implicit arguments always need to be named. To a large degree, the syntactic conciseness of the above examples relies on the powerful combination of implicit function types and inferred eta-expansion. For example, driven by the type annotation, the Dotty compiler automatically eta-expands the body of prog to:

```
implicit (e: Cap[Exc]) ⇒ implicit (a: Cap[Amb]) ⇒ ...
```

... bringing the capabilities for two effects implicitly into scope without having to explicitly bind them. With the capabilities in scope, driven by their types, the compiler will then automatically provide the implicit arguments to calls of raise and flip. Implicit function types are not essential for our approach, since we could always pass down the capabilities by hand. For instance, the example from the introduction can be rewritten to not use implicit at all:

```
val prog2: Cap[Amb] ⇒ Cap[Exc] ⇒ C[Boolean] = amb ⇒ exc ⇒ if (x < 0) flip(amb) else raise("too big")(exc)
```

However, as becomes evident in the example, implicit and implicit function types significantly improve the syntactic convenience.

3 Extensibility Properties

Algebraic effects are well-known for their modularity benefits when compared to other effect structuring techniques such as monads and monad transformers. The effect signature serves as a stable interface between the effect user and the effect implementor. Type class centric MTL style [Liang et al. 1995] offers similar interfacing, but typically semantics to effects in MTL are either provided monolithically or via monad transformers that contradict modularity due to a quadratic number of liftings which have to be written.

Effekt is based on a polymorphic (shallow) embedding [Hofer et al. 2008; Hudak 1998] of effect handlers and thus, like many other algebraic effect libraries, has a solution to the expression problem [Wadler 1998] at its foundation. The EP in context of algebraic effects can be summarized as modularly being able to a) add new operations to an existing effect signature and b) implement new handlers for that effect signature. Effekt inherits the extensibility properties of a polymorphic embedding. The analogy to the EP, however, is not perfect as there are some notable differences: Firstly, effect signatures are degenerate algebraic signatures since the shape of recursion is very regular. In particular, Op never occurs in a contravariant position. Secondly, most descriptions of the EP only consider a single algebra, whereas with algebraic effects we typically have more than one effect signature and the order of handling / folding over the operations affects the semantics.

Additionally, by embedding algebraic effects into a general purpose programming language like Scala, the modularity features of the host language become available to structure effectful programs. From an extensibility point-of-view, Effekt is thus located in the sweet spot of the force field opened up by algebraic effects, the embedding in Scala as a host language and polymorphic embedding as a solution to the EP.

3.1 Dimensions of Extensibility

The remainder of this section relates extensibility dimensions discussed in literature on the expression problem to the algebraic effects setting and shows how Effekt supports them.

Adding new handlers for an effect. The first dimension of the EP. A central feature of every implementation of effects and handlers is the ability to define a new handler for an existing effect. We support this feature: one can define a new trait that implements an existing effect signature.

Adding new operations to an effect. The second dimension of the EP. It is important to distinguish adding an operation to an existing effect and adding a new effect. Effekt supports both in a modular way as required for solutions to the EP. While section 2 already illustrated the latter, let’s look at an example of the former by extending the effect signature of Amb with a nondeterministic choice operator:

```
trait AmbChoose extends Amb {
    def choose[A](choices: List[A]): Op[A]
}
```

We’d like to point out that this introduces a subtyping relationship between Amb and AmbChoose – programs that use the Amb effect can also be handled by a handler supporting AmbChoose. If we want to implement a handler for
AmbChoose we extend AmbList by mixing in AmbChoose and implement the new operation choose.

```scala
trait AmbChooseList[R] extends AmbList[R] with AmbChoose {
  def choose[A](as: List[A]): Op[A] = ...
}
```

For a detailed description of this extensibility dimension in the context of the EP, we refer the interested reader to literature on object algebras [Oliveira and Cook 2012; Oliveira et al. 2013].

### Combining independently developed effect signatures and handlers

The description of the EP has seen many extensions and additional requirements. One additional requirement described by Odersky and Zenger state that the programmer should be able to combine independently developed extensions [2005]. This requirement might seem unnecessary in the context of algebraic effects since instead of combining two effect signatures a user can just use both effects separately. However, this requirement becomes relevant when considering the combination of effect handlers. If we want to handle two effects by interpreting them into the same domain it is in general not enough to run two separately developed handlers in sequence.

```scala
trait ExcList[R] extends Exc with Handler[R, List[R]] {
  def raise[A](msg: String) = pure(List.empty)
}

trait ExcAmbList[R] extends ExcList[R] with AmbList[R]
val res3: C[List[Boolean]] =
  handle(new ExcAmbList[Boolean] | |) { prog }
```

Using ExcAmbList, we can handle both Exc and Amb simultaneously in one handler to yield res3. This is different from sequentially handling prog with AmbList and ExcList, which would result in a value of type List[List[Boolean]]. The example illustrates that in Effekt handlers can be combined with mixin composition under the condition that they interpret the effects into the same domain. By capability subtyping, the combined handler can be used to handle either one of both effects. It is passed down twice, once for each effect it handles.

### Handling effects locally

Being able to handle a subset of the effects used by a program locally is one of the major benefits of algebraic effects with handlers. We have already seen some examples in section 2. One particularly interesting feature is that handlers can again use a set of algebraic effects in their implementation. In Effekt we can express such a dependency on other effects as follows:

```scala
trait AmbFail[R] extends Amb with Handler[R, R] {
  implicit val exc: Cap[Exc]
  def flip() = raise("too drunk to flip.")
  def unit = identity
}

def AmbFail[R](f: R using Amb): R using Exc =
  handle(new AmbFail[R] (val exc = implicitly))(f)
val res4: C[Option[Boolean]] = Maybe { AmbFail | prog }]
```

When compared to EP literature this forwarding to another handler is reminiscent of family self references in [Oliveira et al. 2013] and of base algebras in [Hofer et al. 2008].

### 4 Discussion

This section discusses relevant properties of Effekt’s design.

#### 4.1 Design Decisions

In recent years an impressive amount of research has gone into algebraic effects and handlers [Plotkin and Power 2003; Plotkin and Pretorius 2009]. Programming languages centered around algebraic effects and handlers such as Eff [Bauer and Pretorius 2015], Koka [Leijen 2014] and Frank [Lindley et al. 2017] have been proposed and implemented.

Other research focused on embedding algebraic effects and handlers into existing programming languages. One popular embedding technique is via variations of a free monad [Kammar et al. 2013; Kiselyov and Ishii 2015; Kiselyov et al. 2013] over an open union of effect operations [Swierstra 2008].

#### Multi-prompt delimited continuations

Another, less common, implementation strategy is via multi-prompt delimited continuations. Handling an effect introduces a new prompt marker and this prompt marker is passed down to the call of an effect operation which then captures the continuation up to the prompt. When the host programming language allows for capturing the current context (via call/cc, resumable exceptions, setjmp, etc.) a direct embedding of algebraic effects and handlers is possible [Kammar et al. 2013; Kiselyov and Sivaramakrishnan 2016; Leijen 2017a].

#### Monadic vs. direct style

Since the JVM and in consequence Scala does not support sufficient stack inspection we base our library on a monadic implementation of multi-prompt delimited continuations [Dyvbig et al. 2007]. User programs of our library have to be written in monadic style as can for instance be seen in the handler implementation AmbList.flip above. While for-comprehensions provide some relief, the monadic style might still prevent a wider adoption.

This could be addressed by adding the necessary support for multi-prompt delimited control operators [Kiselyov 2012] to the JVM. We conjecture based on Leijen’s implementation of algebraic effects in C [2017a], that it is possible to implement a direct style variant of Effekt in Scala native. A second approach to support direct style would be to employ a type directed CPS transformation [Rompf et al. 2009].

#### Shallow vs. deep embedding of handlers

Effect implementations based on a deep embedding of effect handlers (this includes all free monad based implementations that we are aware of) reify effectful programs as data. Handlers use pattern matching to interpret the reified programs [Kiselyov and Ishii 2015; Kiselyov and Sivaramakrishnan 2016]. A less
common alternative is a shallow embedding of handlers. In a shallow embedding the semantics is passed down to the part of the program that uses it. In that sense our embedding of effects is shallow: we do not reify an effectful program as a command-response tree, but pass the handler code down to where it is used.

**Capability passing.** We need to pass two things: the prompt marker and the handler code. We call a pair of the two a capability, which is the constructive proof that we indeed have the capability to perform the effect. This is in spirit closer to Frank [Lindley et al. 2017] where we add abilities to an ambient ability than to Koka [Leijen 2014] where we list all effects used by a program but leave some effects open via effect polymorphism. Effect systems based on capabilities offer an alternative perspective on effect polymorphism [Liu 2016; Osvald et al. 2016]. Using implicit parameters for capability passing has been suggested a number of times [Haller and Loiko 2016; Odersky et al. 2017; Osvald et al. 2016].

### 4.2 Safety

In Effekt a capability is only obtainable with a call to handle. A call to use captures the current continuation up to the prompt contained in the capability. For this operation to be safe, it is important that use only occurs in the dynamic scope of the corresponding call to handle. This means a capability should not leak, e.g. via mutable state or capture in a closure. When using such a leaked capability outside of the dynamic scope of handle, the stack maintained by Effekt will not contain the corresponding prompt marker anymore, resulting in a runtime exception.

The version of Effekt as presented in this paper does not address this threat to safety and relies on disciplined use of the library. To guarantee safety and assert that all effects are handled, other implementations of algebraic effects (like the Scala library Eff) use open typelevel unions. This technique could also be applied in our case. However, since we already pass down capabilities, we believe that our library design is well suited to be combined with approaches for second class capabilities [Osvald et al. 2016]. With Dotty’s implicit function types and scala-escape’s @Local annotation we could change the type alias using to be

``` scala
type using[A,E] = implicit @Local Cap[E] ⇒ C[A]
```

expressing that the capability Cap[E] should not be closed over. We combined a Scala version of our library with "scala-escape" which gave first promising results. We hope that our combined use case of implicit function types and @Local to implement safe algebraic effects can guide future research of the Scala language.

Another source of runtime exceptions is a call to run before all effects are handled. This is a standard problem [Dyvbig et al. 2007] and can be addressed by adding enough polymorphism. The signature of run could be changed to be

``` scala
def run[A](f : { def apply(e : Effekt) : e.C[A] }) : A
```

Since the type C is path dependent, capabilities obtained by one instance of Effekt cannot be used in another instance. Since run is not part of the interface of Effekt we can be sure that all effects are handled.

### 4.3 Performance

To evaluate performance, we implemented three different functions count, count8 and nqueens in Effekt and in the established Scala library “Eff”. count only counts down using a simple state effect, count8 does the same but additionally layers eight levels of state effects over a single flip. nqueens is an effect library benchmark from the literature [Kammar et al. 2013]. Running the benchmarks on a 2.5 GHz Intel Core i7 with 16GB of memory results in speed ups of 5.7 – 6.5x, 17.3 – 18.9x and 1.6 – 3.0x, correspondingly. We account the speedup to our use of a shallow embedding, which prevents a heap allocated representation of the effect tree. Also the construction and deconstruction of the effect tree (in terms of pattern matching) is not necessary anymore, potentially allowing for more JIT optimization.

### Conclusion

In this paper, we presented Effekt, a library design for algebraic effects and handlers in Dotty. We described novel extensibility scenarios, which are supported by using a polymorphic embedding of handlers. Building our library around capability passing reduces the need for advanced typelevel programming, such as open typelevel unions.

### Acknowledgments

We would like to thank the anonymous reviewers for their comments that helped improve the paper.

### References


