JFP **30**, e8, 46 pages, 2020.

doi:10.1017/S0956796820000027

Effekt: Capability-passing style for type- and effect-safe, extensible effect handlers in Scala

JONATHAN IMMANUEL BRACHTHÄUSER, PHILIPP SCHUSTER and KLAUS OSTERMANN

University of Tübingen, Tübingen, Germany

(*e-mails:* jonathan.brachthaeuser@uni-tuebingen.de, philipp.schuster@uni-tuebingen.de, klaus.ostermann@uni-tuebingen.de)

Abstract

Effect handlers are a promising way to structure effectful programs in a modular way. We present the Scala library Effekt, which is centered around capability passing and implemented in terms of a monad for multi-prompt delimited continuations. Effekt is the first library implementation of effect handlers that supports effect safety and effect polymorphism without resorting to type-level programming. We describe a novel way of achieving effect safety using intersection types and path-dependent types. The effect system of our library design fits well into the programming paradigm of capability passing and is inspired by the effect system of Zhang & Myers (2019, *Proc. ACM Program. Lang.* **3**(POPL), 5:1-5:29). Capabilities carry an abstract type member, which represents an individual effect type and reflects the use of the capability on the type level. We represent effect rows as the contravariant intersection of effect types. Handlers introduce capabilities and remove components of the intersection type. Reusing the existing type system of Scala, we get effect subtyping and effect polymorphism for free.

1 Introduction

To get a first impression of Effekt, consider the following piece of code written in Scala and using our library. It models a coin toss, but with a twist: the gambler might be too drunk and lose the coin (Kammar *et al.*, 2013). The program uses two *effect operations*: `flip` and `raise`. The effect operation `flip` is used to nondeterministically decide whether the gambler is too drunk to catch the coin. In that case, we use the effect operation `raise` to signal an exception. Otherwise, we return the result of a second coin toss as a string.

```
def drunkFlip(amb: Amb, exc: Exc) = for {
  caught ← amb.flip()
  heads  ← if (caught) amb.flip() else exc.raise("We dropped the coin")
} yield if (heads) "Heads" else "Tails"
```

This simple example already shows a few things. Firstly, the program is written in *monadic style* using Scala's `for`-comprehensions.[1] Even though the program uses multiple effects, all effectful code only uses *one* monad—a variant of the continuation monad. Secondly, the effect operations are methods on *capabilities* `amb` and `exc`, which `drunkFlip` receives as arguments. The semantics of the effect operations is thus dependent on the corresponding implementations of `Amb` and `Exc`. We can, for instance, run the method `drunkFlip` with the handlers `maybe` and `collect`:

```
val res1: List[Option[String]] = run {
  collect { amb ⇒ maybe { exc ⇒ drunkFlip(amb, exc) } }
}
▶ List(Some(Heads), Some(Tails), None)
```

The `collect` handler enumerates all possible outcomes of the `flip` operation and collects them in a list. The `maybe` handler returns `None` if the program raises an exception.

Swapping the two handlers changes the result type and the semantics:

```
val res2: Option[List[String]] = run {
  maybe { exc ⇒ collect { amb ⇒ drunkFlip(amb, exc) } }
}
▶ None
```

This illustrates an important feature of effect handlers. Programs that use effects are agnostic of the concrete handlers and their order (Plotkin & Pretnar, 2009). This gives the caller of the program and the implementer of the handlers more flexibility. Moreover, effect handlers are powerful enough to express many different control-flow abstractions as libraries, which otherwise have to be built into a language. Examples are async-await, cooperative multitasking, iterators, exceptions, and many more (Wu *et al.*, 2014; Leijen, 2016, 2017b; Dolan *et al.*, 2017). We encounter some of these abstractions in the remainder of this paper.

1.1 Effekt - *Effect handlers and capability passing*

In this paper, we present Effekt: a library for programming with effect handlers in the language Scala. The combination of effect handlers with object-oriented features enables new modularization strategies, both for effectful programs and for effect handler implementations. While Scala conveniently provides us with all necessary features, the core ideas behind Effekt (passing effect handlers explicitly as capabilities, combining effect handlers with object-oriented programming, and using path-dependent types and intersection types for effect safety) are independent of our embedding into Scala. The different aspects of our library design are summarized in the type signature of the method `drunkFlip` that we have seen above:

[1] Similar to Haskell's `do`-notation, `for`-comprehensions in Scala syntactically simplify writing monadic code. In general, `for { x1 ← e1; x2 ← e2; ... xn ← en } yield e` desugars to `e1.flatMap { x1 ⇒ e2.flatMap { x2 ⇒ ... en.map { xn ⇒ e }}}`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Like previous versions of Effekt (Brachthäuser & Schuster, 2017), our library design centers around the concept of *capability passing*. As we will see in the remainder of this paper, capabilities in Effekt encapsulate three different things:

Capabilities *contain* Effect Implementations. They give semantics to effect operations (Section 2). In the example program `drunkFlip`, we call effect operations as methods on the capabilities `amb` and `exc`, for instance.

Capabilities *contain* Scope Delimiters. Effect operations can capture the continuation delimited by the corresponding handler. Programs written with our library have type `Control` (Section 3), a monadic implementation of delimited control with first-class prompts (Dybvig *et al.*, 2007). Prompts act as markers and are used to delimit the scope of continuation capture. Capabilities can close over scope delimiters.

Capabilities *contain* Effect Labels. Our capabilities contain a type member (`amb.effect`) that we use as a label on the type level to guarantee effect safety (Section 4). We keep track of all used capabilities by aggregating their effect members in an intersection type as the second-type parameter of `Control`.

1.2 Effect system

Previous versions of Effekt lacked effect safety (Brachthäuser & Schuster, 2017). Capabilities could be used outside of the scope of their handler region, which resulted in runtime exceptions. In this paper, we solve the problem of effect safety and present a variant of Effekt that statically prevents programs from using leaked capabilities. We describe a novel way of achieving effect safety in a library embedding using intersection types and path-dependent types. The effect system of our library design fits well into the programming paradigm of capability passing and is inspired by the effect system of the $\lambda_{\Downarrow\Uparrow}$-calculus by Zhang & Myers (2019). As we will see in Section 4, we represent effect rows as the contravariant intersection of effect types, where an individual effect of a capability `c` is represented by the abstract type member `c.effect`. Handlers remove components of the intersection type. By reusing the existing type system of Scala, we obtain the features of effect subtyping and effect polymorphism for free. By embedding Effekt into a practical, but unsound host language (Amin & Tate, 2016), type and effect safety of our library can only be guaranteed up to soundness of the host language. We do not present formal proofs, but a design for a library implementation as an embedding into an existing, mainstream programming language. Nevertheless, from our experience of working with Effekt, we are confident that it inherits the following properties from $\lambda_{\Downarrow\Uparrow}$.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Effect Safety. Our effect system asserts that all effects are handled. For example, we can only call `run` on a program when all effects are handled and we reject programs like `run { amb.flip() }`. Like with systems based on monadic regions (Launchbury & Sabry, 1997; Moggi & Sabry, 2001; Kiselyov & Shan, 2008), capabilities can *leave* their defining handler scope but our effect system ensures that they cannot be *used* outside their defining handler scope.

Effect Subtyping. We use Scala's support for subtyping of intersection types to implement effect subtyping. A program with type `Control[Int, exc.effect]` can be used where a program of type `Control[Int, exc.effect & amb.effect]` is expected.

Effect Polymorphism. We use Scala's support for type polymorphism to express effect polymorphic functions like:

```
def mapM[A, B, E](ls: List[A], f: A ⇒ Control[B, E]): Control[List[B], E]
```

The function maps an effectful functions over a list. It is polymorphic in the effects `E` used by function `f`. The effects of `mapM` are precisely the effects of `f`. Also, `mapM` may itself use effects and handle them. They are encapsulated and do not appear in the signature.

Existing implementations of languages with effect handlers either completely lack a static effect system—this includes Multicore OCaml (Dolan *et al.*, 2014), Eff (Bauer & Pretnar, 2015), embeddings of Eff in OCaml (Kiselyov & Sivaramakrishnan, 2016), and previous versions of Effekt (Brachthäuser & Schuster, 2017; Brachthäuser *et al.*, 2018)—or they do not have sufficient support for effect polymorphism (Kammar *et al.*, 2013; Inostroza & van der Storm, 2018). Languages and libraries with effect systems like Extensible Effects (Kiselyov *et al.*, 2013), Koka (Leijen, 2017c), Links (Hillerström *et al.*, 2017), Frank (Lindley *et al.*, 2017), and Helium (Biernacki *et al.*, 2019) require explicit lifting annotations to *encapsulate effects* in effectful higher-order functions, like the function `mapM` above. Without such manual liftings, the implementation detail of effects used within `mapM` would leak into its type signature. In contrast, Effekt requires no such manual lifting.

Related to effect encapsulation is the property of *effect parametricity* (Zhang & Myers, 2019). Just by looking at the type of `mapM` above, we can guarantee that no implementation of `mapM` can (accidentally or purposefully) handle effects `E` used by `f`. Handling an effect means two things: delimiting captured continuations and providing an implementation for effect operations. By employing capability passing, Effekt guarantees that no matter how `mapM` is implemented—it will never change the implementations of effect operations used in `f`. However, it is possible for `mapM` to instantiate delimit continuations for effects `E`, which are captured within `f`. This breaks effect parametricity as we explain in Section 5.4 in more detail.

1.3 List of contributions and paper overview

In summary, this article makes the following contributions:

- We introduce the library design of Effekt, which is based on implementing effects in terms of other effects. In particular, continuations are captured using the `Scope`

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

effect, which serves as a base case. The corresponding built-in effect handler for `Scope` delimits the extent of the captured continuation.

- We implement Effekt building on the operational semantics of Dybvig *et al.* (2007). We achieve effect safety by generalizing techniques of Launchbury & Sabry (1997) to nested regions (Kiselyov & Shan, 2008) and using intersection types of abstract type members (Parreaux *et al.*, 2017) instead of rank-2 types. Our effect system rules out some use cases of multi-prompt delimited control (Kobori *et al.*, 2016), but can express many interesting use cases of effect handlers.
- We present our API design of *ambient state* (Kiselyov *et al.*, 2006; Leijen, 2018) as the second of the two built-in effects: the `State` effect.
- We evaluate the usability of our library in multiple extended case studies.
- We evaluate desirable properties like effect safety and effect parametricity of our library embedding and discuss limitations.
- We evaluate the extensibility properties of our library design. We discuss interesting opportunities to explore type- and effect-safe modularization of effectful programs, opened up by embedding Effekt into Scala, a language that combines functional programming with object-oriented programming.

The remainder of the paper is structured as follows. In Section 2, we give an overview of programming with Effekt and show how to implement effect handlers. While ultimately we aim to achieve an effect-safe implementation of effect handlers, as an intermediate step, we present a monadic implementation of delimited control (Section 3) to then show how to restrict it to an effect-safe, yet useful subset (Section 4). We discuss properties and limitations of our effect system in Section 5. In Section 6, we give several extended case studies and discuss novel extensibility properties that arise from our embedding into Scala. Section 7 discusses related work and Section 8 concludes.

2 Programming with effect handlers in Effekt

To introduce programming with effect handlers in our library Effekt, we continue to use the effects from the introduction as a running example. We will see how to declare and handle the exception and ambiguity effects. This section should give the reader a high-level intuition for the usage of the library. The examples of this section have been presented in similar form in Koka (Leijen, 2017c) and previous versions of Effekt for Scala (2017) and Java (2018). All code is given in Dotty (version 0.19), the upcoming version of the Scala programming language. The library and examples from this paper are available online:

`https://github.com/b-studios/scala-effekt/tree/jfp`

Programming with effect handlers encourages modularity by separating the interface of an effect (the effect signature) from its implementation (the effect handler). This is reminiscent of how programs are organized in object-oriented programming. In fact, embedding effect handlers into Scala—a language with support for functional as well as object-oriented paradigms—we actively use the correspondence between effect handlers and object-oriented programming (summarized in Figure 1).

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Effect Handlers	Object-Oriented Programming
Effect Signatures	Interfaces
Effect Operation	Method
Effect Handlers	Implementations / Classes
Effectful Programs	Interface Users
Effect Capability	Instances / Objects

Fig. 1. Mapping concepts from effect handlers to object-oriented programming.

2.1 Example: Exceptions—aborting the computation

Exceptions are a simple effect and provide a good opportunity to introduce the involved concepts. Our effect system guarantees that all effects are handled, which means that the exceptions we implement are checked. This is not the case for Scala's native exceptions they are unchecked.

Scala Background. In Scala, traits not only have value members, but also type members. Value members and type members can be left abstract and defined in implementing classes (Odersky & Zenger, 2005b). The syntax `x ⇒ EXPR` introduces a lambda that binds `x` in `EXPR`. Methods with one argument (like `map`) can be written infix (i.e., `c map f` desugars to `c.map(f)`). Methods with only one argument can be called with braces, instead of parenthesis, that is, `c.map(f)` and `c.map { f }` are equivalent for the purpose of this paper.

2.1.1 Effect signatures are interfaces

Effect signatures group multiple effect operations under one type. In Effekt, we represent effect signatures as interfaces (called "traits" in Scala). Figure 2(a) declares the effect signature `Exc`. The return type of effect operation `raise` makes use of the type alias:

```
type /[+Result, -Effects] = Control[Result, Effects]
```

In Scala, type constructors with two arguments can be used infix and we write `R / E` to denote the type `Control[R, E]`. We read `R / E` as the type of a computation with result type `R`, using effects `E`. The variance annotations `+` and `-` make it explicit that effectful computations are *covariant* in their result type and *contravariant* in their effects. The return type of effect operations like `raise` not only tells us the type of the result, in this case `Nothing` (the bottom of Scala's subtyping lattice). It also describes which effects an effect operation may use, in this case the abstract type member `effect` (which is short for `this.effect`). The type member `effect` is declared in the library trait `Eff`, that `Exc` inherits from:

```
trait Eff { type effect }
```

We only use effect types like `effect` for effect safety. They do not have any operational meaning attached to them and are erased by the compiler.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Eff { type effect }
trait Exc extends Eff { def raise(msg: String): Nothing / effect }
trait Amb extends Eff { def flip(): Boolean / effect }
```

(a) Effect signatures for exception and ambiguity as traits extending library trait `Eff`.

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E =
  // (1) handlers introduce a new scope
  handle { scope ⇒
    // (2) create handler instance / capability
    val exc = new Exc {
      // (3) communicate the use of ‘scope‘
      type effect = scope.effect
      // (4) implement effect operations
      def raise(msg: String) = scope.switch { resume ⇒ pure(None) }
    }
    // (5) provide handled program with capability
    prog(exc) map { r ⇒ Some(r) }
    //        ^^^ (6) lift pure values into the effect domain
  }
```

(b) Handler function for the exception effect.

```
def collect[R, E](prog: (amb: Amb) ⇒ R / (amb.effect & E)): List[R] / E =
  handle { scope ⇒
    val amb = new Amb {
      type effect = scope.effect
      def flip() = scope.switch { resume ⇒
        for { xs ← resume(true); ys ← resume(false) } yield xs ++ ys
      }
    }
    prog(amb) map { r ⇒ List(r) }
  }
```

(c) Handler function for the ambiguity effect.

Fig. 2. Using Effekt to declare and handle exception and ambiguity effects.

2.1.2 Capabilities are effect instances

Consider the following effectful function which uses a capability for the exception effect[2]:

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

We read the type signature of `div` as “given an exception capability `exc`, `div` computes an integer using the capability `exc`”. While mentioning `exc` twice in the type signature seems redundant, Effekt separates two concerns often conflated in existing effect languages.

[2] To syntactically separate capabilities from other arguments, in the example, we curry the function definitions using multiple argument sections.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Dynamic Effect Semantics. Passing `exc` as parameter gives the program (term-level) access to the methods of `Exc`, in this case the effect operation `raise`. Other languages with support for effect handlers perform a search at runtime to handle effect operations like `raise`. In contrast, using Effekt, we explicitly pass effect handlers in the form of capabilities as parameters.

Static Effect Semantics. When we use the effect operations of the `exc` capability to implement `div`, we have to mention the type member `exc.effect` in the effect type of `div`. We will go into the details of the effect system in Section 4—for now it is enough to understand that we guarantee effect safety by tracking all unhandled effects in the type parameter `Effects` of `Control`, that is, the right-hand side of `R / E`. Most of the time, the return type of effectful functions like `div` can be inferred.

2.1.3 Effect handlers are implementations

The effect signature `Exc` only specifies the available effect operations. To give them a concrete interpretation, we define a *handler function* with the following signature:

```
def maybe[R,E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E
```

Handler functions fulfill two purposes. Firstly, they provide capabilities (i.e., `exc`) as arguments to the handled program. Secondly, handler functions remove the used effect (i.e., `exc.effect`) from the effect type. The type of `prog` is a path-dependent function type. The intersection type (i.e., `exc.effect & E`) reflects that `prog` might use the `exc` capability passed to it, as well as other effects `E`. The handler function runs the program `prog` and effectively removes `exc.effect` so the final effect type is just `E`. The handler is polymorphic in both—in the result type `R` of the handled program `prog` and in all other effects `E` that the program might use, and which are not handled by `maybe`.

Operationally, the handler function interprets the effectful program which would compute a result of type `R` (mnemonic for "result type") into a new semantic domain of type `Option[R]`, the *effect domain*. As seen earlier, programs that raise exceptions will be handled to return `None`. Programs that do not raise an exception return `Some(result)`.

Remark. We interchangeably use the terminology *capability* and *handler instance*. While "capability" puts a focus on the concept of entitling the holder to use an effect, "handler instance" highlights the fact that handlers are implementations of effect signatures.

2.1.4 Handler implementations can shift the perspective

Figure 2(b) uses the Effekt library to implement the handler function `maybe`. In Effekt, effect handlers can use other effects to implement effect operations. Often, handlers need to capture a *delimited continuation*. This is achieved using the built-in effect `Scope`, which declares the effect operation `switch` (Section 3). The `Scope` effect, in turn, can be handled with the built-in library function `handle`. It introduces a new capability, which we bind to `scope` in this example. Continuations captured by `scope.switch` will be delimited by this very call to `handle`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

The anonymous handler instance `exc` extends the effect signature `Exc` and implements the method `raise`. It uses `scope.switch` to capture the current continuation. We communicate the use of `scope` on the type level by defining the type member `effect` to be `scope.effect`. Since we call `handle` at type `Option[R] / E`, the method `scope.switch` has type:

```
def switch[A](body: (A ⇒ Option[R] / E) ⇒ Option[R] / E): A / scope.effect
//                  ^^^^^^^^^^^^^^^^^^^
//                   the continuation
```

Calling `scope.switch` in our implementation of `raise` switches the current evaluation context: it transfers the control flow from the call site of `raise` to the respective (dynamically enclosing) call to `handle`. Conceptually, the body passed to `switch` is thus evaluated at the call to `handle` not at the call to `raise`. Calling `switch` also captures the current continuation and provides it as the argument (i.e., `resume`). The continuation allows us to transfer the control flow back to the original call site of `raise`. In this example, however, we discard the continuation and immediately return `None`. Operationally, the call to `handle { ... }` will thus be replaced by `pure(None)`, aborting the computation. The captured continuation `resume` corresponds to the context between the call to `handle { scope ⇒ ... }` and the call to `scope.switch`. While at runtime `prog` might arbitrarily install delimiters before calling `raise`, the connection between the two functions `scope.switch` and `handle` is *statically scoped*. We know that for all calls to `exc.raise` the captured continuation will be delimited by this lexically enclosing `handle`, even in the presence of duplicate instances of this delimiter as we discuss in Section 5.4.1.

2.1.5 Return clauses

Delimiting the scope with `handle` requires the result type of the passed program to match the effect domain of the handler—that is `Option[R]` in our case. For this reason, we are mapping over the result of the program `prog` to wrap it in the constructor `Some`. In other languages like Koka, Eff, or Frank, this lifting of the result type into the effect domain is typically performed by *return clauses*. Using capability passing, return clauses are not required to be part of the user interface of handlers while maintaining the same expressive power (Section 5.5).

2.2 Example: Ambiguity—resuming multiple times

Our implementation of the exception effect discards the continuation of the program when it encounters a `raise` and immediately returns `None`. In contrast, the handler for ambiguity illustrates how to use the continuation to resume to the original call site. Figure 2(a) declares the effect signature of the `Amb` effect. The implementation of the handler function `collect` in Figure 2(c) handles the `Amb` effect. It changes the result type of the program from `R` to the effect domain `List[R]`:

```
def collect[R, E](prog: (amb: Amb) ⇒ R / (amb.effect & E)): List[R]
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

To implement the `flip` operation, we capture the continuation and call it with the result of the coin flip. Here, the type of the continuation is `resume: Boolean ⇒ List[R] / E`, so calling it with either `true` or `false` gives us a list of possible results `List[R]`. To explore all execution paths we call `resume` twice, once with `true` and once with `false`. Finally, we concatenate both lists with `xs ++ ys`. Like before, we lift the result type `R` into the effect domain (this time `List[R]`) by wrapping the result in a singleton list. That is, if there is no call to `flip`, no ambiguity arises and the list contains only one result.

2.3 The Effekt library

In this section, we encountered the basic concepts of programming with effect handlers in Effekt. While all effectful computation happens in one monad (`Control`), programming with effect handlers encourages a modularization into three components: effect signatures, effectful programs, and effect handlers.

Effect signatures like `Exc` are interfaces containing methods marked as effectful with an abstract type member `effect`. This abstraction is very powerful—not only is the implementation of the method left abstract, but we also leave open which effects an implementation might use. In a concrete implementation, all effectful methods share the type member `effect` much like all methods of an object share the private state.

Effectful programs use effect operations by calling into explicitly passed capabilities. This has advantages, but also can be a burden due to its verbosity. Like we did in previous versions of Effekt for Scala (Brachthäuser & Schuster, 2017), we could hide most of it using implicit parameters and implicit function types. However, for most parts of this paper, we refrain from doing so to reduce cognitive overhead and focus on the aspect of effect safety. Section 6.3.1 discusses the use of implicits in greater detail.

Effect handlers provide semantics to effect operations. We distinguish three different aspects of an effect handler. The *handler function*, like `collect`, is a higher-order function that provides an `amb` capability and removes the `amb.effect` from the effect type of the handled program. The *handler implementation* is a class implementing the effect signature. In our above example, the `Amb` interface is implemented by an anonymous inner class `new Amb { ... }`. The *handler instance*, like `amb`, is an instance of the handler implementation. Handler functions encapsulate three aspects of effect handling in one module:

1. the handler function uses `handle` to delimit the scope of the captured continuation;
2. it locally uses the fresh scope introduced by `handle` to implement the effect operations in terms of `scope.switch`—the effect operations close over `scope` and are thus the only way to capture the continuation;
3. it finally lifts the return type of the handled function `R` into the effect domain which makes it the answer type of the delimiter `handle`.

Grouping these aspects of effect handling in one module, it is possible to locally reason about type safety. The implementation of `raise` is only safe because we statically know that the answer type expected at `handle` is `Option[R]`. Likewise, in `collect`, we know that the answer type in the body of `scope.switch` is `List[R]`. This allows us to safely concatenate the results of the two calls of the continuation `resume`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

The remainder of this paper iterates the running example of this section and introduces all mentioned types and library functions in three steps: Section 3 gives an overview over the underlying implementation of delimited control. Section 4 then establishes effect safety for this implementation of delimited control and introduces the abstraction of effect signatures. Finally, Section 6 explores newly gained extensibility by refactoring the handler implementations of `maybe` and `collect` into reusable and extensible traits.

3 Answer-type safe effect handlers

Effekt implements effect handlers in terms of a monad for delimited control. In previous sections, we used a monad `Control[+Result, -Effects]` that is both answer-type safe and effect-safe. To focus on the operational semantics of delimited control and to illustrate problems of effect safety, in this section, we start with a simpler variant `Control[+Result]` which has the same operational semantics is answer-type safe, but not effect-safe. It roughly corresponds to the one presented in earlier work (Brachthäuser & Schuster, 2017). In Section 4, we show how to achieve effect safety.

3.1 Structured programming with delimited continuations

For multiple decades, control operators like `call/cc` have been used to program with control effects (Friedman *et al.*, 1984). Recently, in disguise, control effects have found their way into mainstream programming languages in the form of specialized solutions such as `async`/`await`, fibers, coroutines, generators, and others.

Also recently, the programming languages research community found new interest in control effects in the form of algebraic effects (Plotkin & Power, 2003) and their extension with handlers (Plotkin & Pretnar, 2009, 2013). Effect handlers occupy a sweet spot between the general control operators (such as `call/cc`) and specialized programming language features (such as `async`/`await`). Like general control operators, effect handlers are very powerful and can express many of the above language features as user-defined libraries (Dolan *et al.*, 2015, 2017; Leijen, 2017b). However, unlike general control operators, effect handlers also encourage modularity. We believe the regained interest comes from four important improvements over control operators like `call/cc`:

1. generalizing from undelimited to delimited continuations
2. generalizing from one control operator to a family of control operators
3. establishing answer-type safety of control operators
4. establishing effect safety of control operators

From an engineer's perspective, each of these improvements helps to write programs in a modular way making them easier to extend and making it easier to reason about parts of a program in isolation.

The connection between effect handlers and delimited control is not accidental. It has been established practically (Kiselyov & Sivaramakrishnan, 2018) as well as theoretically (Forster *et al.*, 2017; Piróg *et al.*, 2019) that certain forms of delimited continuations can express certain forms of algebraic effect handlers. Similarly, in the literature, effect handlers are sometimes introduced as a structured way to programming with delimited continuations (Kammar *et al.*, 2013; Leijen, 2017c).

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Control[+Result] {
  def flatMap[B](f: Result ⇒ Control[B]): Control[B]
  def map[B](f: Result ⇒ B): Control[B]
  def andThen[B](c: Control[B]): Control[B] = flatMap { res ⇒ c }
}
def pure[A](value: A): Control[A] = ...
def run[A](prog: Control[A]): A = ...
```

(a) The monad `Control` for programming with delimited control.

```
trait Eff {}
trait Scope[R] extends Eff {
  def switch[A](body: (A ⇒ Control[R]) ⇒ Control[R]): Control[A]
}
def handle[R](prog: (scope: Scope[R]) ⇒ Control[R]): Control[R] = ...
```

(b) The `Scope` effect with effect operation `switch`, and built-in handler function `handle`.

Fig. 3. The API of Effekt without effect typing.

> *"Effect handlers are to delimited continuations as structured programming is to goto"* — attributed to Andrej Bauer by Kammar et al. (2013).

In the design of the Effekt interface (presented in Figure 3), we take this quote literally. In particular, there are two ways to view the monad `Control`.

An Embedding of Delimited Control. One can view `Control` as an embedding of a language with (delimited) control effects into Scala. The library function `handle` is a delimiter for the control effects in the provided program `prog`. It also introduces a fresh `Scope` that can be thought of as a *prompt marker* (Sitaram & Felleisen, 1990; Gunter *et al.*, 1995) labeling the corresponding call to `handle`. The method `switch` is a control operator. It captures (and removes) the current continuation up to (and including) the corresponding `handle`. It then passes the continuation to the provided body. Taking this point of view, programming with `Control` as presented in the introduction is "just" structured programming with delimited continuations.

Delimited Control as Built-in Effect. For the purpose of this paper, however, we want to take an alternative point of view. Staying in the conceptual framework of effect handlers, we think of `switch` as an effect operation, defined in the effect signature `Scope`. The function `handle` is a handler function for the `Scope` effect. Effect handlers can be defined in terms of other effects, including the `Scope` effect. The only difference between the `Scope` effect and user-defined effects like `Amb` or `Exc` is that the handler function `handle` is built into the `Control` monad. This point of view emphasizes programming with effects and handlers. The potential use of the `Scope` effect to capture the continuation is an implementation detail of the respective user-defined handler.

3.2 The `Control` monad—Delimiting continuations

Figure 3(a) defines the interface of our monad for delimited control. The implementation of our monad `Control[+Result]` is based on the monad for multi-prompt delimited control

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

by Dybvig *et al.* (2007). We specialize both, syntax and semantics, to better fit effect handlers. As usual, we embed a pure value into the monad with `pure` and we sequence effectful computations with `flatMap`. This enables us to write effectful programs in an imperative style via Scala's for-comprehensions. Using `run[A]`, we execute a program with control effects that computes a value of type `A`. The type parameter `Result` of our monad is marked as covariant (i.e., `+Result`) to enable correct subtyping.

3.2.1 Capturing delimited continuations with `Scope.switch`

Besides being a monad, `Control` enables the implementation of one particular built-in effect: `Scope`. Figure 3(b) defines the `Scope` effect and its corresponding built-in handler `handle`. The effect operation `switch` captures the continuation and binds it to `resume`. It is important to emphasize that the continuation is delimited. Such a *delimited continuation* (Felleisen, 1988) is also referred to in literature as subcontinuation (Hieb *et al.*, 1994) or partial continuation (Johnson & Duggan, 1988). Since their introduction by Felleisen, many variants of delimited control operators have been discovered—maybe the most prominent example being shift / reset (Danvy & Filinski, 1992). Using shift / reset, we can express the following example:

$$\mathsf{print}(\mathsf{reset}\{\ (1 + \ \mathsf{shift}\{k \Rightarrow k(3) + k(4)\}\ ) * 2\ \})$$

The continuation captured by shift (highlighted in gray) corresponds to the evaluation context $(1 + \square) * 2$. Its extent is *delimited* by reset and thus does not include the call to print. The example will print the number $((1 + 3) * 2) + ((1 + 4) * 2) = 18$. Importantly, since it is delimited, calling the continuation $k(3)$ *does* return with the value 8.

Example 1

We can translate the above example to Scala using the `Control` monad and the `Scope` effect:

```
val ex1: Control[Unit] = handle { scope ⇒
  scope.switch { resume ⇒ for {
    x ← resume(3)
    y ← resume(4)
  } yield x + y } map { x ⇒ (1 + x) * 2 }
} map { x ⇒ print(x) }
```

Like above, running `run { ex1 }` will print the number `18`. Adjusting the example illustrates the choice of terminology for `scope.switch`.

```
val ex1b: Control[Unit] = handle { scope ⇒
  scope.switch { resume ⇒ pure(42) } map { x ⇒ (1 + x) * 2 }
} map { x ⇒ print(x) }
```

The call to `switch` changes (i.e., "switches") the context to the `handle` call that introduced the scope. The body passed to `switch` is evaluated in the context print(□). Returning from the body *is* returning from `handle` and running `run { ex1b }` prints `42`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

3.2.2 Multiple scopes and families of control operators

We will now see how to use the `Control` monad to program with multiple scopes (Sitaram & Felleisen, 1990; Gunter *et al.*, 1995). Every call to `handle` introduces a fresh `scope` that corresponds to a prompt marker (Sitaram & Felleisen, 1990). This gives rise to a dynamic number of control operators `scope.switch`, one for each call to `handle`. The following example illustrates the use of multiple delimiters.

Example 2

We use `handle` twice, introducing two different scopes s_1 and s_2.

```
val ex2: Control[Int] = handle { s1 ⇒
  handle { s2 ⇒
    s1.switch { resume ⇒ pure(21) }
  } map { x ⇒ if (x) 1 else 2 }
} map { x ⇒ 2 * x }
```

The captured continuation `resume` contains the program segment delimited by scope s_1, that is, the evaluation context $\mathsf{handle}\,\{\,s_1 \Rightarrow \mathsf{if}\,(\mathsf{handle}\,\{\,s_2 \Rightarrow \square\,\})\,1\,\mathsf{else}\,2\,\}$. Here, the body of `switch` discards the continuation and returns `21`. Hence, `run { ex2 }` gives `42`.

3.2.3 Answer-type safety

Operationally, `scope.switch { k ⇒ PROG }` replaces the corresponding call to `handle` by the body `PROG`. The return type of the body has to match the answer type at the `handle`. To guarantee this, we follow Gunter *et al.* (1995) and parametrize the `Scope` effect by the answer-type `R` (Figure 3(b)). In Example 2, the two scope capabilities thus have types `s1: Scope[Int]` and `s2: Scope[Boolean]`. The type of `handle[R]` in Figure 3(b) requires three types to be `R`: the answer type of the created scope (i.e., `Scope[R]`), the result of the given program (i.e., `Control[R]`), and the return type of `handle`. Similarly, the type of `switch` uses the answer type of the given scope and requires that (a) the return type of the continuation and (b) the return type of the given body agree with (c) the answer type expected at the `handle` that introduced this `scope`. Answer-type safety is especially important for multiple nested scopes. Each call to `handle` might introduce a scope with a different answer type. Switching to a handler with the wrong type should be statically rejected. For instance, switching to s_2 instead of s_1 would render Example 2 type-incorrect. It would require the body of `switch` to return a computation of type `Boolean`, not `Int`.

4 Effect-safe effect handlers

In the previous section, we have seen a version of `Control` that has a type parameter `Result`. Also indexing `Scope` with `Result`, we statically track answer types and guarantee that capturing and calling the continuation is type-safe. However, this version of `Control` is not effect-safe: capabilities can leave the scope of the handler function that introduced them, that is, the handler *region*. Using a `Scope` capability outside of the handler

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

region leads to a runtime error. We illustrate two ways for capabilities to leave the handler region.

Leaving the Handler Region by Returning. We can leave the handler region by returning from it. In this case, it is possible to leak the `Scope` capability either through references

```
var s: Scope[Unit] = null
val problem1 = run {
  for {
    _ ← handle { scope ⇒ s = scope ; pure(()) }
    _ ← s.switch { resume ⇒ pure(()) } // Exception: undelimited scope
  } yield ()
}
```

or simply by returning it as result:

```
val problem2 = run {
  for {
    s ← handle { scope ⇒ pure( scope ) }
    _ ← s.switch { resume ⇒ pure(()) } // Exception: undelimited scope
  } yield ()
}
```

As also observed by Osvald *et al.* (2016), both sources of leakage can occur indirectly through functions or objects that close over the capability. Capabilities might even leave the scope of the enclosing `run` to then be used in the scope of a different `run`. Dybvig *et al.* (2007) use rank-2 types to prevent this particular source of runtime errors, but leave others to future work.

Leaving the Handler Region by Scope Switching. We can also leave the scope of handle by means of control effects, that is, by switching the scope.

```
val problem3 = run {
  handle { scope ⇒
    scope.switch { resume ⇒
      scope.switch { resume ⇒ pure(()) } // Exception: undelimited scope
    }
  }
}
```

Since switching the scope removes the enclosing handle, switching a second time inside of the body of `switch` results in a runtime error. The evaluation context of the second switch is run { □} and the captured continuation is not delimited anymore. Danvy & Filinski (1990) operationally prevent this kind of runtime error by leaving the outer delimiter behind. That is, the evaluation context would correspond to run { handle { *scope* ⇒ □} }. However, in

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

the setting of multiple prompts/scopes this is not sufficient. The delimiter can be removed by switching to a different scope:

```
val problem4 = run {
  handle { s1 ⇒ handle { s2 ⇒
    s1.switch { resume ⇒
      s2.switch { resume ⇒ pure(()) } // Exception: undelimited scope
    }
  }}
}
```

4.1 Establishing effect safety

We now introduce our implementation of an effect system that rules out the above four problematic programs, prevents the use of escaped capabilities, and guarantees effect safety. The underlying problem that our effect system solves is a general one: we need to restrict the lifetime of a resource (capabilities in our case) to a certain dynamic region (the call to `handle` in our case). This problem occurs in the domain of region-based resource management (Kiselyov & Shan, 2008), object capabilities (Haller & Loiko, 2016), delimited control (Dybvig *et al.*, 2007), scope safety in type-safe meta-programming (Parreaux *et al.*, 2017), as well as with prompt-based implementations of effect handlers (Brachthäuser & Schuster, 2017). Our effect system is inspired by the $\lambda_{\Downarrow\Uparrow}$ calculus (Zhang & Myers, 2019), which uses dependent types to track the set of used labels (i.e., capabilities) in the effect type. In Section 5, we discuss some properties of our effect system embedding, but leave formal proofs of safety and soundness to future work. For a better comparison, Appendix A gives a more immediate embedding of $\lambda_{\Downarrow\Uparrow}$ into Scala.

4.2 Tracking capabilities

Following Zhang & Myers (2019), our effect system builds on the idea of tracking the set of capabilities used by a program in the type of the program. To enable tracking of effects, Figure 4 thus defines our final version of `Control` with a second-type parameter `Effects`. We represent capabilities (i.e., instances of type `Eff`) on the type level by their abstract type member `effect` and we use Scala's intersection types to describe a set of capabilities.

As an example, assuming capabilities c_1, c_2, and c_3 we write the type of the effectful program that uses c_1 and c_2 to compute an integer as:

```
val prog1: Control[Int, c1.effect & c2.effect]
```

Here, `c1.effect` is the abstract type member of the capability and `c1.effect & c2.effect` is an intersection type. The intersection might be uninhabited, but this is irrelevant for our use case. We only use the intersection type as a phantom type to track used effects.

To support effect subtyping, the type parameter `Effects` of `Control` is marked as contravariant (i.e., `-Effects`).

```
val prog2: Control[Int, c1.effect & c2.effect & c3.effect] = prog1
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
type Pure = Any
type /[+R, -E] = Control[R, E]
```

(a) Type aliases `Pure` to describe the empty set of effects and `/` for infix notation.

```
trait Control[+Result, -Effects] {
  def flatMap[B, E](f: Result ⇒ B / E): B / (Effects & E)
  def map[B](f: Result ⇒ B): B / Effects
  def andThen[B, E](c: B / E): B / (Effects & E) = flatMap { prog ⇒ c }
}
def pure[A](value: A): A / Pure = ...
def run[A](program: A / Pure): A = ...
```

(b) The monad `Control` for programming with delimited control.

```
trait Eff { type effect }
trait Scope[R, E] extends Eff {
  def switch[A](body: (A ⇒ R / E) ⇒ R / E): A / effect
}
def handle[R, E](prog: (s: Scope[R, E]) ⇒ R / (s.effect & E)): R / E = ...
```

(c) The `Scope` effect with effect operation `switch`, and built-in handler function `handle`.

Fig. 4. The API of Effekt with effect typing.

The above assignment is type correct, since we have by subtyping:

```
c1.effect & c2.effect & c3.effect <: c1.effect & c2.effect
```

We model the *syntactically* empty intersection of capability types by defining the type alias `type Pure = Any`, where `Any` is the top of the Scala subtyping lattice. Pure programs have an effect type `Pure` since they do not use any capabilities. By contravariance, they are a subtype of effectful programs that do contain capability types in the intersection. It is important to point out that the purity only refers to delimited control. Programs with a pure effect row still can use side effects like writing to files or accessing the network.

Every use of a `Scope` capability, for example, `scope.switch`, taints the effect type of its result. The return type `A / effect` of `switch` (Figure 4) indicates the use of `switch` on this capability on the type level. The type of `run` asserts that only pure programs without control effects can be executed. That is, all capabilities have to be removed from the set of effects and it has to be `Pure`. To guarantee safety, we have to make sure that the only way to remove an effect from the intersection is by delimiting the program with `handle`:

```
def handle[R, E](prog: (s: Scope[R, E]) ⇒ R / (s.effect & E)): R / E
```

Here, `prog` has a *dependent function type*: the return type is (path)-dependent on its value parameter `s`. Importantly, in our implementation, we leave the type member `effect` of `s` abstract. Different calls to `handle` lead to different types `s.effect` that cannot be unified by the type checker. Hence, only the call to `handle` that introduced a scope capability can remove its very own abstract effect type from the set of effects. This excludes problematic programs like $\texttt{problem}_1$ and $\texttt{problem}_2$ from above.

Our approach to achieve effect safety is similar to how rank-2 types can be used to enable type-safe monadic regions in Haskell (Launchbury & Sabry, 1997; Moggi & Sabry, 2001;

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Kiselyov & Shan, 2008). Using path-dependent types, we move the universal quantification from the type level to the term level. Since rank-2 types are not well supported in Scala, using abstract type members also improves ergonomics and type inference.

Remark. In Scala, two (path-dependent) types are equal if and only if their prefix paths are stable and they can be unified (Odersky & Zenger, 2005b). Informally, a path is stable if it does not contain a mutable component. This way we prevent leakage via mutable references as in $\texttt{problem}_1$.

4.3 From answer-type safety to effect safety

In the previous variant of `Control`, scopes carried the answer-type `R` to ensure that using control effects is type-safe. Crucially, to establish effect safety and prevent programs like $\texttt{problem}_4$ from type checking, the type `Scope` (Figure 4) now has an additional type parameter `E`.

```
trait Scope[R, E]
```

With this change, scope capabilities now track both the expected return type `R` and the set of capabilities `E` available at the corresponding handler. The type `Scope[R, E]` can conceptually be understood as type `Scope[R / E]`. However, we track the two aspects in separate type parameters to improve type inference.

Intuitively, the body of a `switch` is evaluated at the position of the corresponding call to `handle`. This is also reflected in the type of the body:

```
body: (A ⇒ R / E) ⇒ R / E
```

Both, the answer-type `R` and the effects `E` have to match the corresponding `handle[R, E]`. Thus, capability-passing style is not only essential for operationally delimiting control effects but also necessary to carry both the expected answer type as well as the available effects from the definition site `handle { scope ⇒ ... }` down to the use site `scope.switch`.

Since the body of `switch` has to return `R / E`, it cannot shift to the same scope again (as in $\texttt{problem}_3$). This would require a type of `R / (scope.effect & E)`. The problematic program $\texttt{problem}_4$ is ruled out too: $\texttt{s}_1$ has type `Scope[Int, Pure]` and thus the body of the first `switch` needs to be pure and cannot use $\texttt{s}_2$.

Example 1 —Effect typed

We are now ready to revisit the examples from the previous section and assign effect types. On the term level, the example is the same, but its typing is more precise:

```
val ex: Unit / Pure = handle { (scope: Scope[Int, Pure]) ⇒
  scope.switch { resume ⇒ for {
    x ← resume(3)
    y ← resume(4)
  } yield x + y } map { x ⇒ (1 + x) * 2 }
} map { x ⇒ print(x) }
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

The effect type of `ex` shows that after handling there are no more undelimited control effects left and we can safely run the program.

Example 2 —Effect typed

The second example illustrates how each handle removes its corresponding scope capability from the effect type.

```
val ex2: Int / Pure = handle { (s1: Scope[Int, Pure])⇒
  handle { (s2: Scope[Boolean, s1.effect]) ⇒
    s1.switch { resume ⇒ pure(21) } // Control[Int, s2.effect & s1.effect]
  } map { x ⇒ if (x) 1 else 2 }     // Control[Int, s1.effect]
} map { x ⇒ 2 * x }                 // Control[Int, Pure]
```

The body of the second handle has type `Control[Int, s1.effect & s2.effect]`. By effect subtyping, we can still use `s1.switch {...}`, which has type `Control[Int, s1.effect]`.

The example also highlights an important aspect of our effect-safe control operator: scope capabilities track the available effects at the *definition site*, not the *use site*. For instance, the type of `s2` informs us that within a body of `s2.switch { ... }`, the `s1.effect` can be used. In contrast, the return type of `s2.switch` in this example would only mention `s2.effect`, not `s1.effect`. From the point of view of effect handlers, the potential usage of `s1.effect` in the body of `s2.switch { ... }` is an implementation detail that is encapsulated at the definition site. It does not leak to the use site.

Coin flipping example – Effect typed

The handler function `maybe` from Section 2 implements a user-defined effect `Exc` in terms of another effect `Scope`. Let us recall the effect signature of `Exc`

```
trait Exc extends Eff { def raise(msg: String): Nothing / effect }
```

and the (type annotated) `maybe` handler function:

```
def maybe[R, E](prog: (exc: Exc) ⇒ R / (exc.effect & E)): Option[R] / E =
  handle { scope ⇒
    val exc: Exc { type effect = scope.effect } = new Exc {
      type effect = scope.effect
      def raise(msg: String) = scope.switch { resume ⇒ pure(None) }
    }
    (prog(exc): R / (scope.effect & E)) map { r ⇒ Some(r) }
  }
```

The implementation `exc` of `Exc` uses the `scope` capability. The type of `exc` is therefore refined from `Exc` to `Exc { type effect = scope.effect }`. This allows the type checker to locally unify `exc.effect` and `scope.effect`. It is also necessary for `handle` to remove the `scope.effect` component of the effect type, resulting in the return type `Option[R] / E`. Since the type refinement is local, the use of the capability `scope` in the

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

implementation of `exc` and the type equivalence between `exc.effect` and `scope.effect` remains unknown to the handled program `prog`. This implementation detail is encapsulated in the `maybe` handler function. Similarly, as we will see in Section 6, also the usage of effects other than `Scope` can be encapsulated.

Having introduced the effect system, we are ready to type- and effect-check the example program from the introduction:

```
val res1 = run {
  collect { amb ⇒
    maybe { exc ⇒
      drunkFlip(amb, exc)    // Control[String, exc.effect & amb.effect]
    }                        // Control[Option[String], amb.effect]
  }                          // Control[List[Option[String]], Pure]
}                            // List[Option[String]]
```

Each handler function removes an effect and at the same time changes the effect domain.

4.4 Effect-safe ambient state

Many practical handler implementations require some form of state. Our host language Scala already supports mutable state and effect handlers can readily use it. However, combining mutable state and delimited control can interact in unforeseen ways (Kiselyov *et al.*, 2006; Leijen, 2018). This is illustrated by the example in Figure 5(a). In the program on the left, we introduce a local mutable variable `x`. We only modify it if the `flip` operation returns `true`. Still, running the example outputs `List(2, 2)`. Surprisingly, the change to `x` is also visible in the branch where `flip` returns `false`. Built-in mutable state is *global* and does not backtrack across different resumptions.

For some effect handlers, we want *local*, backtrackable state. Running the example program should return `List(2, 0)`, backtracking the local state when resuming the second time. Leijen (2017c) calls this form of state "ambient". There are multiple ways to achieve this desired behavior. One way is to define a state effect in terms of the `Scope` effect. For effect handlers, this technique has been presented by Kammar *et al.* (2013), and for delimited control by Kiselyov *et al.* (2006). Another way would be to offer a generalized form of effect handlers that support local state. For example, Koka (Leijen, 2017c) supports "parametrized handlers" which manually perform state-passing.

4.4.1 The `State` effect

For our design of Effekt, we explore a third approach and offer a built-in `State` effect that exhibits the correct backtracking behavior in combination with the `Scope` effect. Figure 5(b) defines the interface of the effect signature `State` and the corresponding built-in handler `region`. The use of the state effect is illustrated in the right column of Figure 5(a). Given a capability `state`, we create a new field `x` with `val x = state.Field(...)`. To read and update the state, we use the effect operations `get` and `put` on the field. Our types make sure that we can only access a field within the region of the state capability that we used to create the field.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
collect { amb ⇒
  var x = 0
  amb.flip() map { b ⇒
    if (b) { x = 2 } else {}
    x
  }
}
▶ List(2, 2)
```

```
collect { amb ⇒ region { state ⇒
  val x = state.Field(0)
   amb.flip() flatMap { b ⇒
     if (b) x.put(2) else pure(())
  } andThen x.get()
}}

▶ List(2, 0)
```

(a) Example illustrating the difference between global (left) and local, backtrackable state (right).

```
trait State extends Eff {
  def Field[T](value: T): Field[T]

  trait Field[T] {
    def get(): T / effect
    def put(value: T): Unit / effect
    def update(f: T ⇒ T): Unit / effect = get() map f flatMap put
  }
}
def region[R, E](prog: (s: State) ⇒ R / (s.effect & E)): R / E = ...
```

(b) Effect signature of the state effect and its built-in handler `region`.

Fig. 5. Effect-safe state effect.

The type of the nested trait `Field` looks like an effect signature. However, instead of extending `Eff`, it refers to the type member `effect` on the outer trait `State`. This way, all fields created by calling the method `Field` share the same effect (Leijen 2018 also refers to this parent effect as "umbrella effect"). This way a dynamic number of fields can be created, while maintaining the invariant that fields cannot escape the region of the corresponding state handler.

Our implementation of `Control` is specialized to properly save and restore the fields for each `State` effect. The `region` handler introduces a new mutable frame which holds the allocated fields on the stack that the implementation of `Control` is based on. Capturing the continuation with `scope` will capture parts of this stack, shallowly copying the current values of the fields. Calling the continuation restores the values. This way, access and modification are possible in constant time, at the cost of (shallow) copying fields on continuation capture. If we would switch the order of the effect handlers to

```
region { state ⇒ collect { amb ⇒ ... }}
```

running the example would again yield `List(2, 2)`. In this order, changes to the state are persisted across multiple resumptions. In Section 6, we will encounter additional examples using the built-in state effect.

5 Properties of the effect system

Our effect system is based on the $\lambda_{\Downarrow\Uparrow}$ calculus by Zhang & Myers (2019). We embed their calculus into the practical programming language Scala, which has no full formal specification and will therefore not formally prove properties of our library. Nevertheless,

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

in this section, we discuss some properties of our embedded effect system and explain under which assumptions we believe them to hold.

5.1 Effect safety

Effect safety is the absence of runtime errors, caused by capabilities being used outside of their defining handlers. Assuming a sound subset of Scala, such as the pDot calculus (Rapoport & Lhoták, 2019), we are confident that our effect system establishes effect safety—though we do not give formal proofs. Adding mutable variables and fields to the calculus should also not affect our effect system, which relies on *stable*, that is, immutable paths. The same also holds for native exceptions, though our library as presented in this paper is not prepared to interact with native exceptions. A formal treatment is left to future work. In our experience, adding effect types to existing advanced case studies (such as the `Scheduler` handler in Section 7) helped us to discover a few subtle bugs. The effect system also guided us in the design of the interface for ambient state as presented in Section 4.4.

5.2 Effect subtyping

By marking the set of capabilities in `Control` as contravariant, we use Scala's support for subtyping of intersection types to express effect subtyping. For example, a program only using the `amb` effect is a subtype of a program using `amb` and `exc`:

```
Control[R, amb.effect] <: Control[R, amb.effect & exc.effect]
```

As explained in Section 4, this is the case since we have

```
amb.effect & exc.effect <: amb.effect
```

Reusing Scala's subtyping for effect subtyping is an important advantage over effect systems that encode effect rows using type-level lists. In those systems, effect subtyping typically has to be implemented manually by performing type-level computation (Kiselyov & Ishii, 2015). In contrast, using intersection types to express the set of effects integrates well with other Scala features like variance annotations and type bounds. Type inference for monotonically growing intersection types is well supported, and in consequence most return types of effectful functions (like `drunkFlip`) and handlers (like `maybe`) can be omitted.

5.3 Effect polymorphism

Since effect types are Scala types, we can also reuse Scala's support for type polymorphism to express effect polymorphic functions. One example of an effect polymorphic, higher-order function is

```
def mapM[A, B, E](lst: List[A], f: A ⇒ B / E): List[B] / E
```

The function `mapM` is effect polymorphic in the effects `E` used by function `f`. The return type of `mapM` indicates that it potentially calls `f` in its implementation and so has the same

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

effects as `f`. The effects `E` still need to be handled by the caller of `mapM`. Handler functions like `collect` and `maybe` (Section 2) are other examples for effect polymorphic functions. Type inference for calling higher-order functions (like `mapM`) that do not alter the set of effects is well supported. In contrast, nesting multiple handler applications (like `collect` and `maybe`) often requires users to explicitly list the remaining effects by provide typing annotations.

5.4 Effect parametricity

The example function `mapM` is polymorphic in the effects `E`. Following Zhang & Myers (2019), we claim that it *should not be possible* for the implementation of `mapM` to (accidentally) handle any concrete effect in `E`; no matter what `E` will be instantiated to at the call site. That is, in the following user program, we should be able to determine *statically* that `flip` is handled by `collect`. No implementation of `mapM` should be able to violate this assumption.

```
collect { amb ⇒ mapM(List(1,2,3), n ⇒ amb.flip()) }
```

We refer to this property as *effect parametricity*. It has also been called *abstraction safety* in the literature (Zhang & Myers, 2019). Generally speaking, given a type, parametricity allows us to infer properties of the runtime behavior. In the case of effect parametricity, we want to guarantee the absence of accidental handling. But does effect parametricity hold for our implementation of Effekt? The answer is more subtle than in most other implementations of effects and handlers because Effekt is based on capability passing. We need to distinguish two different aspects of accidental handling:

Implementation Parametricity. Accidental handling of effects can occur in languages without static effect systems like Eff (Bauer & Pretnar, 2015) and Multicore OCaml (Dolan *et al.*, 2014). Those languages dynamically search handlers at runtime. In those systems, `mapM` could (accidentally or purposefully) handle `Amb` and, for example, change the semantics of the `flip` operation to always return `true`. Previous presentations of Effekt (Brachthäuser & Schuster, 2017; Brachthäuser *et al.*, 2018) did not have an effect system. Still, they already support this aspect of effect parametricity. Just like presented in this paper, user programs are written in capability-passing style. Capabilities are passed down to their use site and not looked up at runtime. In our example, the function passed to `mapM` closes over the capability `amb`, which fixes the implementation of `flip`.

Scope Parametricity. Because Effekt uses an implementation of multi-prompt delimited continuations under the hood, there is a second aspect of effect parametricity to consider. Looking at the example call to `mapM` again, we would also like to be sure that the continuation captured by `flip` will always be delimited by the corresponding call to `collect` and nowhere else. Because `mapM` does not know about `amb`, it should not be possible for `mapM` to install a scope delimiter for `amb`. In Effekt, we can construct examples that violate this property as we will see next.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

5.4.1 Capturing the scope delimiter by leaking the continuation

Every call to `handle` creates a fresh scope. Surprisingly, it is possible to delimit a program given an existing scope, even if we do not offer this in the API of Effekt.[3] The following example illustrates (accidental) delimiting of continuations.

```
handle[String, Pure] { s ⇒
  delimit(s) {
    s.switch { resume ⇒ pure("abort") }
  } map { x ⇒ "Did not abort" }
}
```

We use `handle` to delimit a program and create a new scope `s`. Within the delimiter, we then use `s.switch` to abort the current computation and return the string `"abort"` as the overall result. We now want to statically know that the continuation captured by `s.switch` is delimited by the call to `handle` that created `s`, independent of the implementation of `delimit`. In particular, we might think that running the program will always return the string `"abort"`, given that `delimit` forces the computation passed to it. In Effekt, this does not hold. Maybe surprisingly, we can write a function `delimit`, such that the example program returns the string `"Did not abort"` instead of aborting to the outer scope.

Installing Delimiters by Resuming. The continuation captured by `s.switch` contains the delimiter for `s` and calling the continuation reinstalls the delimiter. This fact is not represented in the type of `resume`. It only mentions the effects it *uses* not the ones it *delimits*. Using this insight, we can define `delimit`:

```
def delimit[R,E](s: Scope[R,E])(prog: R / s.effect): R / (s.effect & E) =
  s.switch[R / (s.effect & E)] { resume ⇒
    resume( resume(prog) )
  }.flatMap(c ⇒ c)
```

The implementation of `delimit` captures the continuation `resume`, which contains the delimiter for scope `s`. The call to `s.switch` returns a *computation* that will call the continuation on the given program. We force this computation *outside* of the call to `s.switch` with `flatmap(c ⇒ c)`. This effectively duplicates the captured context and the delimiter for `s`, as illustrated by the following reduction:

$$
\begin{aligned}
\mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\mathsf{delimit}(s)\,\{\,\mathit{body}\,\}]\,\} &\longmapsto \mathsf{resume}(\lambda().\,\mathsf{resume}(\mathit{body})) \\
&\longmapsto \mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\,(\lambda().\,\mathsf{resume}(\mathit{body}))()\,]\,\} \\
&\longmapsto \mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\,\mathsf{resume}(\mathit{body})\,]\,\} \\
&\longmapsto \mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\,\mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\,\mathit{body}()\,]\,\}\,]\,\} \\
&\quad \text{where } \mathsf{resume} = \lambda c.\,\mathsf{handle}\,\{\,s \Rightarrow \mathsf{E}[\,c()\,]\,\}
\end{aligned}
$$

Here, we use E to denote the evaluation context between the call to `delimit` and the corresponding handler for `s` (Wright & Felleisen, 1994). Evaluating the call to `delimit` reifies the context and binds it to `resume`. Forcing the passed continuation (e.g., $c()$) is part of this captured context. Now, calling the continuation reinstalls the evaluation context between

[3] We are grateful to an anonymous reviewer, who pointed out this source of violating parametricity.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

the call to `delimit` and (including) the original call to `handle`. Calling it again, after the context has already been restored, duplicates the context. If *body* now captures and discards the continuation, only the innermost copy of the context handle { $s \Rightarrow \mathsf{E}[\square]$ } is removed. Hence, our example returns `"Did not abort"`.

Importantly, the captured context E can contain *arbitrary other* handler frames for scopes $s \neq s'$. That is, all other delimiters in the evaluation context `E` are duplicated, even though the method `delimit` neither has access to them nor mentions them in its type signature.

5.4.2 Discussion

As can be seen from the example, violating scope parametricity requires some careful engineering. While it is necessary for the continuation to leave the effect operation that captured it, this is not sufficient. It is important to note that just calling the continuation twice (e.g., `resume(); resume()`) does not cause the capturing behavior. When the continuation is called for the second time, the delimiters reinstalled by the first call have already been removed. The same holds for executing code *after* calling a continuation (e.g., `resume(); f()`). Effects in `f` are *not* captured by the delimiters in `resume`. To observe accidental capture, the executed code has to be part of the continuation itself. In our example, we use a higher-order function and pass the code to be executed to the continuation.

In our experience in working with the library, we did not find the lost scope parametricity to be a problem in practice. To instantiate multiple copies of the continuation required us to use higher-order, effect polymorphic effect operations. Other effect handler implementations come with different linguistic restrictions, ruling out this particular capturing scenario. For example, Koka and the $\lambda_{\Downarrow\Uparrow}$ calculus do not support effect polymorphic effect operations, which makes it difficult to express `delimit`. Scala Effekt is a practical implementation, embedded in Scala. Using Scala's type polymorphism to model effect polymorphism, effect operations can naturally also make use of this feature. It is not clear to us which restrictions are desirable and practically enforceable without ruling out useful programs (like the ones in the following section).

Finally, we are not aware of any effect handler language, where the type of the continuation reflects the effects it handles. Depending on the other features of the particular language, it thus might still be possible to construct a scenario similar to the one described above. We leave it to future work to further investigate the problem and explore solutions, both for stand-alone languages, as well as for embeddings like Effekt.

5.5 Effect encapsulation

Another property, *effect encapsulation*, is related to effect parametricity, and violations of it can be observed in languages featuring an ML-like type system with row polymorphism for effect types like Koka and Frank (Lindley, 2018; Leijen, 2018). The following program is adapted from Leijen (2018) and written in the Koka language. It shows how an effect used by f_1 leaks into its type—it is not encapsulated.

```
fun f1(action: () → <exc|e> a): e option<a> { // types inferred
  maybe { if (...) { raise("abort") }; action() }
}
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

Here, `f1` is a higher-order function that takes an effectful function `action` as its argument. The function `f1` uses exceptions in its implementation but locally handles them with the `maybe` handler. This implementation detail *leaks* as part of the inferred type which states the fact that `exc` effects of `action` will be handled by `f1`. Koka implements effect subtyping via row polymorphism so the effect row of `action()` is unified with those of other statements handled by `maybe`.

5.5.1 Manually encapsulating effects

Operationally, Koka will handle any exception effect used in `action` with the `maybe` handler in `f1`. We cannot hide this fact by annotating the parameter `action` of `f1` with the type `() → e a`, which does not type-check. However, if we do not want any exceptions thrown by `action` to be handled by `maybe`, languages like Koka and Frank offer some form of manual lifting operation (Biernacki *et al.*, 2017; Convent *et al.*, 2019).

```
fun f2(action: () → e a): e option<a> { // types inferred
  maybe {
    if (...) { raise("abort") }
    inject<exc> { action() }
  }
}
```

Manually injecting the `exc` effect into the effect row also has operational content as described by Leijen (2018): the runtime search for the exception handler will skip the `maybe` handler in `f2`. Now the type of `f2` truthfully states that it does not handle any effects in `e`—including any exception effects.

In Effekt, we can directly express the two variants of the function `f` with different types:

```
def f1[A, E](action: (exc: Exc) ⇒ A / (exc.effect & E)): Option[A] / E
def f2[A, E](action: () ⇒ A / E): Option[A] / E
```

The type of `f1` makes it clear that `action` has an unhandled exception effect, handled by `f1`. Furthermore, no `inject` is necessary to select the right handler, since capabilities are named and passed explicitly.

5.5.2 Encapsulation in return clauses

In addition to the implementation of effect operations, in languages like Koka, Eff, or Frank handlers also need to implement a *return clause* (called `unit` in the following example):

```
val exc = new Exc {
  def unit(r: R): Option[R] / E = pure(Some(r))
  def raise(msg: String) = scope.switch { resume ⇒ pure(None) }
}
```

This additional abstraction exists both for historical and technical reasons. Historically, algebraic effect handlers were conceived as a fold over the tree of computation operations (Plotkin & Pretnar, 2009). Return clauses are required to lift pure values into the domain of computations. In Effekt, it is possible to express return clauses in terms of mapping over the result like:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
handle { scope ⇒ val exc = ...; prog(exc) flatMap exc.unit }
```

However, languages like Koka require some form of lifting or `inject` to make this transformation since effect operations used in the return clause might accidentally be handled by the same handler that has the return clause (Leijen, 2018; Lindley, 2018). Again, this is not an issue in Effekt, since the connection between handler and operations is established explicitly via capability passing. Using capability passing, return clauses are not required to be part of the user interface of handlers while maintaining the same expressive power.

6 Even more extensible effects

In previous sections, we have seen how to write simple programs with Effekt (Section 2) and how to establish effect safety (Section 4). In this section, we now give additional examples to evaluate the different dimensions of extensibility gained by embedding Effekt into Scala—a language that supports functional and object-oriented paradigms. In particular, we will see how to compose effect signatures, effect handlers, and effectful programs.

6.1 Composing effect signatures

In the introduction, we alluded to the fact that mapping effect signatures and handlers to existing features of object-oriented programming allows us to reuse the abstractions those features offer. In particular, as we will see, mapping signatures to Scala's traits opens up interesting new modularity benefits.

6.1.1 Extending effect signatures

Since signatures are traits, we can extend them and add new operations. Here, we extend the `Amb` trait (defined in Figure 2(a), Section 2) with an additional effect operation `choose`.

```
trait Choose extends Amb {
  def choose[A](first: A, second: A): A / effect
}
```

This introduces a subtyping relationship between `Amb` capabilities and `Choose` capabilities. A handler for `Choose` thus can also be used to handle `Amb`. This cannot be expressed in Koka, for example, where a handler is tied to a single-effect signature and can only handle precisely the effect with this signature.

6.1.2 Default methods: primitive versus derived effect operations

Traits in Scala cannot only contain abstract method declarations, but also concrete method implementations. Similarly, our effect signatures cannot only contain abstract operations, but also concrete effect operation implementations, as illustrated in the following example.

```
trait Fiber extends Eff {
  // primitive effect operations
  def suspend(): Unit / effect
  def fork(): Boolean / effect
  def exit(): Nothing / effect
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
  // derived effect operations
  def forked[E](p: Unit / E): Unit / (E & effect) = for {
    b ← fork()
    r ← if (b) p andThen exit() else pure(())
  } yield r
}
```

Here, the `Fiber` effect (Dolan *et al.*, 2015; Leijen, 2017b) for cooperative multitasking has three abstract effect operations that need to be implemented by handlers. The effect operation `suspend` indicates that the current fiber can be suspended. The effect operation `fork` is similar to the equally named system call in Unix, spawns a new fiber, and returns `true` if the code is executed as part of this new fiber. Finally, the effect operation `exit` terminates the current fiber. In addition to the three abstract effect operations, the effect signature also contains a concrete effect operation `forked`. The argument computation of type `Unit / E` represents the code, which is executed as part of the forked fiber. The effect operation `forked` is implemented in terms of `fork` and `exit`. We refer to operations like `suspend` as *primitive effect operation* and to operations like `forked` as *derived effect operations*. Derived effect operations can be overridden in handler implementations, for example, for efficiency. Furthermore, the derived operation `forked` does not make any assumptions about the handler implementation. In particular, it does not explicitly capture the continuation with `scope` but only uses the other effect operations of `Fiber`. This illustrates an important difference to languages like Koka, where every effect operation always automatically captures the continuation.

6.1.3 Abstract type members: Effect signatures as module interfaces

Another particularly interesting example of abstraction reuse is Scala's abstract type members. Mapping effect signatures to Scala traits, they cannot only describe effect operations, but also have (abstract) type members. This opens up interesting ways to structure effect signatures, illustrated by the following signature of the `Async` effect (Dolan *et al.*, 2017):

```
trait Async extends Eff {
  type Promise[T]
  def async[T, E](prog: T / E): Promise[T] / (E & effect)
  def await[T](p: Promise[T]): T / effect
}
```

The abstract type member `Promise` allows handler implementations to choose the representation of promises. The effect signature also declares two effect operations (`async` and `await`) that refer to the abstract type. The operation `async` takes an effectful computation and returns a promise that can be awaited with the effect operation `await`. In a concrete capability of type `Async`, the choice of representation for `Promise` is hidden existentially. Hiding the representation prevents instances of promises to be awaited outside of the corresponding `Async`-handler.

Example: Asynchronous Programming. Having declared the effect signatures for `Async` and `Fiber` we can write effectful programs using these effects:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
def asyncEx(f: Fiber, a: Async) = for {
    p ← a.async { for {
        _ ← log("Async 1")
        _ ← f.suspend()
        _ ← log("Async 2")
        _ ← f.suspend()
      } yield 42 }
    _ ← log("Main")
    r ← a.await(p)
    _ ← log("Main with result " + r)
  } yield ()
```

We will see how to run this example after having defined the handlers for `Fiber` and `Async`.

6.1.4 Nested traits: Families of effectful types

Abstract type members like `Promise` are not the only way to express a family of effectful types. Traits in Scala can also be nested, and we can refactor the effect signature `Async` to:

```
trait AsyncNested extends Eff {
  trait Promise[T] { def await: T / effect }
  def async[T, E](prog: T/ E): Promise[T] / (E & effect)
}
```

Like in the effect signature of the `State` effect (Figure 5(b), Section 4), the type `effect` used by `Promise` refers to the type member on `AsyncNested`.

6.1.5 Mixing effect signatures

Scala supports mixin composition on traits (Odersky & Zenger, 2005b) so we can mix independently declared effect signatures. For example, we can mix the two effect signatures for ambiguity and exceptions to obtain a combined effect signature for nondeterminism:

```
trait Nondet extends Amb with Exc
```

Furthermore, since traits can contain both abstract and concrete definitions, effect signatures can be mixed to mutually implement primitive (i.e., abstract) effect operations in one signature by derived (i.e., concrete) effect operations in another. This works for both abstract types and abstract methods.

6.2 Composing effect handlers

All effect handlers so far were implemented as anonymous inner classes. This style of implementing handlers precludes certain forms of reuse.

6.2.1 Handlers as traits

To recover extensibility and reuse, we can express effect handlers as traits. Effects that are required by the implementation can be expressed as abstract value members. For instance, expressing the `maybe` handler (Figure 2(b)) as a trait looks like:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Maybe[R, E] extends Exc {
  val scope: Scope[Option[R], E]
  type effect = scope.effect
  def raise(msg: String) = scope.switch { resume ⇒ pure(None) }
}
```

Here we express the dependency on the scope effect as an abstract value member `scope`. The type of `Scope` communicates that it needs to be delimited at some optional type. Otherwise, to increase reuse, the `Maybe` handler is parametric in type `R` and the set of effects `E` at the scope delimiter. Similarly, we can express the handler for ambiguity as a trait:

```
trait Collect[R, E] extends Amb {
  val scope: Scope[List[R], E]
  type effect = scope.effect
  def flip() = scope.switch { resume ⇒ for {
    xs ← resume(true)
    ys ← resume(false)
  } yield xs ++ ys }
}
```

Handling the `Amb` effect with the `Collect` handler trait now amounts to constructing a handler instance of `Collect` and passing it to the program.

```
handle { s ⇒
  val amb = new Collect[R, E] { val scope: s.type = s }
  prog(amb) map { r ⇒ List(r) }
}
```

The highlighted type refinement is essential. It allows the type checker to locally unify `scope.effect`, `s.effect`, and `amb.effect`. This way, `handle` can remove `amb.effect` from the set of effects of `prog`. Expressing handlers as traits turns them into reusable, composable, and extensible components. We will now explore the different dimensions of extensibility enabled by this technique.

6.2.2 *The effect expression problem*

Many implementations of libraries and languages for (algebraic) effects and handlers are based on a *deep embedding* of effect operations. They reify effect operations as alternatives in a sum type and represent effectful computations as a command-response tree. For instance, the `flip` effect operation would be reified as a constructor of an algebraic data type `Amb`. Handlers fold over the tree of computation and interpret the reified effect operations by pattern matching on them (Bauer & Pretnar, 2015; Kiselyov & Ishii, 2015; Hillerström *et al.*, 2017; Leijen, 2017c; Kiselyov & Sivaramakrishnan, 2018). To mix programs with different effects means to extend an open union type of reified effect operations.

In contrast, by performing capability passing and representing effect signatures as traits, Effekt builds on a *shallow embedding* (Hudak, 1998; Carette *et al.*, 2007) of effect operations. Instead of folding over the tree of computation, user programs directly call effect

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

operations on the handler. In a language with mixin composition, shallow embeddings can be structured in a pleasingly extensible way (Oliveira & Cook, 2012). Thus, Effekt has a solution to the *expression problem* (Wadler, 1998) at its foundation, a property it shares with many other effect handler implementations. For instance, languages like Koka (Leijen, 2014), Frank (Lindley *et al.*, 2017), and Links (Hillerström *et al.*, 2017) are based on row polymorphism (Gaster & Jones, 1996) and Extensible Effects (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) are based on open unions (Liang *et al.*, 1995).

We relate extensibility dimensions discussed in the literature on the expression problem to effect handlers (Brachthäuser & Schuster, 2017) and show how Effekt supports them:

Adding New Handlers. The first dimension of the effect expression problem corresponds to adding a new function definition over the recursive data type in the original expression problem. A central feature of every implementation of effects and handlers is the ability to define a new handler for an existing effect. Effekt supports this feature: users can define a new trait or class that implements an existing effect signature.

Adding New Operations. The second dimension of the effect expression problem corresponds to adding a new variant to the recursive data type in the original expression problem. We can distinguish between adding an operation to an existing effect signature and adding a new effect signature. Effekt supports the modular extension of effect signatures as illustrated by the example trait `Choose` of Subsection 6.1. Other languages like Koka cannot compose effect signatures. In those languages, it is therefore also not necessary to compose or extend handler implementations. Effekt allows the programmer to extend handler implementations modularly.

```
trait CollectChoose[R, E] extends Collect[R, E] with Choose {
  def choose[A](first: A, second: A): A / effect = for {
    b ← flip()
  } yield if (b) first else second
}
```

In this example, the handler for the extended effect signature `Choose` extends the existing `Collect` handler and only implements the missing effect operation `choose`. The example also illustrates that we can reuse the implementation of `flip` to implement `choose`.

6.2.3 Mixing handlers—horizontal composition of handlers

The description of the expression problem has seen many extensions and additional requirements. One additional requirement described by Odersky & Zenger (2005a) is that the programmer should be able to combine independently developed extensions. For effect handlers, this means to compose two existing effect handlers. This feature might seem unnecessary in the context of effect handlers where handler composition can be expressed by function composition. However, using trait mixin composition to combine two handlers, the handler implementations can share implementation details like private methods and dependencies on other internally used effects. As an example, we define another handler for ambiguity that performs backtracking to compute only the first successful result:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Backtrack[R, E] extends Amb {
  val scope: Scope[Option[R], E]
  type effect = scope.effect
  def flip() = scope.switch { resume ⇒ for {
    attempt ← resume(true)
    res ← if (attempt.isDefined) pure(attempt) else resume(false)
  } yield res }
}
```

We can implement the `Nondet` effect simply by mixing the handlers `Backtrack` and `Maybe`:

```
def nondet[R,E](prog: (b: Nondet) ⇒ R / (b.effect & E)): Option[R] / E =
  handle { s ⇒
    val n = new Nondet with Backtrack[R,E] with Maybe[R,E] {
      val scope: s.type = s }
    prog(n) map { r ⇒ Some(r) }
  }
```

The use of mixin composition is legal, since the two handlers use the same effects. In particular, both the answer-type `Option[R]` and the set of effects `E` on `scope` coincide. Using the handler function `nondet`, we can handle `Exc` and `Amb` simultaneously:

```
val res3: Option[String] = run { nondet { n ⇒ drunkFlip(n, n) } }
▶ Some("Heads")
```

The example illustrates how handlers can be composed *horizontally* with mixin composition under the condition that they interpret the effects into the same effect domain. Operationally, they share the same scope delimiter. By subtyping, the combined handler can be used to handle both effects. In $\texttt{res}_3$, it is passed down twice, once to handle the `Amb` effect and once to handle the `Exc` effect.

6.2.4 Composition over inheritance—vertical composition of handlers

Effect handlers allow us to locally handle a subset of effects used by a program. To do so, handlers can again use effects in their implementation which are then handled by other handlers. That is, we can compose handlers *vertically*. However, so far this composition was not particularly interesting. All handlers, that we have so far encountered used the `Scope` effect and consequently defined `type effect = scope.effect`.

Different to most other formulations of effect handlers, handlers in Effekt do not have to capture and use the continuation and consequently do not have to use the `Scope` effect. It is up to the handler implementation to decide. Figure 6 presents an example of such a handler that does *not* use the scope effect to capture the continuation. Instead, it uses the effects `State` and `Fiber` and therefore has the capabilities `state` and `fiber` as abstract value members. We use the `Fiber` effect to fork the computation in `async` and to implement polling in `await`. We define `Promise` to be the type `state.Field` and implement `async` to store the result of the asynchronous computation to the field provided by the `state` capability. The handler function `poll` takes the two required capabilities to construct an instance of the handler trait `Poll`:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Poll extends Async {
  val state: State; val fiber: Fiber

  type effect = state.effect & fiber.effect
  type Promise[T] = state.Field[Option[T]]

  def async[T, E](prog: T / E) = for {
    p ← pure(state.Field[Option[T]](None))
    _ ← fiber.forked { prog flatMap { r ⇒ p.put(Some(r)) }
  } yield p

  def await[T](p: Promise[T]) = p.get() flatMap {
    case Some(r) ⇒ pure(r)
    case None ⇒ fiber.suspend() andThen await(p)
  }
}
```

Fig. 6. Handler for the `Async` effect – using two effects `State` and `Fiber`.

```
trait Scheduler[E] extends Fiber {
  val state: State
  val scope: Scope[Unit, state.effect & E]
  type effect = scope.effect

  type Queue = List[Unit / (state.effect & E)]
  lazy val queue = state.Field[Queue](Nil)

  def exit() = scope.switch { resume ⇒ pure(()) }
  def fork() = scope.switch { resume ⇒
    queue.update { resume(true) :: resume(false) :: _ } andThen run
  }
  def suspend() = scope.switch { resume ⇒
    queue.update { _ appended resume(()) } andThen run
  }

  private def run: Unit / (state.effect & E) = queue.get() flatMap {
    case Nil ⇒ pure(())
    case p :: rest ⇒ queue.put(rest) andThen p andThen run
  }
}
```

Fig. 7. Handler for the `Fiber` effect — using the `State` effect *after* switching the scope.

```
def poll[R,E](s: State, f: Fiber)(prog: (a: Async) ⇒ R / (a.effect & E))
  = prog(new Poll { val state: s.type = s; val fiber: f.type = f })
```

By refining the types of `state` and `fiber` to singleton types, the inferred return type of `poll` is `R / (E & s.effect & f.effect)`. It thus communicates precisely that we implement the `Async` effect in terms of the given state and fiber capabilities.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

6.2.5 Vertical composition and scopes

The `Poll` handler (Figure 6) uses two effects `State` and `Fiber` in its implementation, but does not use the `Scope` effect to capture the continuation. All the other effect handlers we have seen so far do use the scope effect (e.g., a capability of type `Scope[R, E]`) but are parametric in the set of outer effects `E`. That is, after switching the scope those handlers did not use any other effects. However, there is another combination of using effects that we have not seen so far: first, switch the scope and *then* use another effect.

Figure 7 uses this technique to implement the `Fiber` effect as a round robin scheduler. It requires the two capabilities `state` and `scope`. It uses the state capability to maintain a queue of fibers that still needs to be scheduled. Fibers are obtained by capturing the continuation using the `scope` capability. The difference to the previous examples now is in the type of the `scope` capability. It informs us that the state effect is available *after* switching the scope. This is not relevant for the implementation of the effect operation `exit` which simply discards the continuation. However, the two remaining operations `fork` and `suspend` both use the state effect in the body passed to `scope.switch`.

The type of the `scope` capability imposes an order on how `state` and `scope` need to be handled. The state effect is required to be the outer handler. Not only is this important to be able to use state after switching the scope, it is also important for the semantics of the scheduler: the handler uses state to store a queue of running fibers. The state should be persisted across different fibers, which are forked by resuming once with `true` and once with `false`. The handler function for the scheduler directly handles the `Scope` effect, but leaves the state effect open. The inferred return type of `scheduler` is thus `Unit / (st.effect & E)`.

```
def scheduler[E](st: State)(prog: (f: Fiber) ⇒ Unit / (f.effect & E)) =
  handle { sc ⇒ prog(new Scheduler[E] {
      val scope: sc.type = sc; val state: st.type = st
  })}
```

This mode of use is how all effect handlers are implemented in Koka, Eff, and Frank. In those languages, the body of an effect operation is always executed at the call site of the handler, not the operation. Also all effects used in the implementation of the effect operation will be evaluated at the handler call site (i.e., the call to `handle` in our case). This is essential to encapsulate effects as implementation details and not leak their usage into the call site. As seen with the `Poll` handler, Effekt offers an alternative to encapsulate effects by employing capability passing and using (path-dependent) abstract type members.

Example: Running Asynchronous Programs. Having defined handlers for the `Fiber` and `Async` effects, we can now finally run `asyncEx` (Subsection 6.1.3) as follows

```
region { s ⇒                                                Async 1
  scheduler[s.effect](s) { f ⇒                              Main
    poll(s, f) { a ⇒                                        Async 2
      asyncEx(f, a) // Unit / (f.effect & a.effect)         Main with result 42
    }               // Unit / (f.effect & s.effect)
  }                 // Unit / s.effect
}                   // Unit / Pure
```

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

resulting in the output on the right. Type inference is not proficient enough to infer the removal of effects and we need to annotate the effect type `s.effect` at the call to `scheduler`.

6.3 Composing effectful programs

By passing capabilities explicitly, we are able to select which instance of an effect to use in the presence of multiple instances of the same effect. For example, we can use multiple instances of `Fiber` in one program to model different thread pools. At the same time, explicit capability passing can be a burden since it introduces manual boilerplate. This is illustrated by the following example that uses the three effects ambiguity, exceptions, and reading from an input stream to model a parser. The parser recognizes an arbitrary number of characters `'a'` followed by a single character `'b'`.

```
// AB ::= a AB | b
def parseAB(amb: Amb, exc: Exc, in: Input) = alternative(
  accept('a')(in, exc) andThen parseAB(amb, exc, in) map { _ + 1 },
  accept('b')(in, exc) map { x ⇒ 0 })(amb)

def accept(exp: Char)(in: Input, exc: Exc) = in.read() flatMap { t ⇒
  if (t == exp) pure(()) else exc.raise("Expected " + exp) }

def alternative[A, E](fst: A / E, snd: A / E)(amb: Amb) =
  amb.flip() flatMap { b ⇒ if (b) fst else snd }
```

In the example, we use an `Input` effect that supports reading from an input stream.

```
trait Input extends Eff { def read(): Char / effect }
```

When composing effectful programs that use different effects, the programmer needs to manually pass the capabilities to the respective function calls. In particular, a function like `parseAB` that uses other effectful functions (like `accept` and `alternative`) needs to take the *union* of all capabilities required by its dependencies. To call them, the programmer needs to select the correct subset of capabilities and provide them along other arguments. For example, all three capabilities need to be passed to the recursive call.

6.3.1 Implicits for capability-passing style

While the overall design of Effekt is largely independent of Scala, there are certain features that ease the use of the library. One such feature is implicit parameters. Implicit parameters (now called "given-clauses" or "contextual parameters" in Scala 3) can help to automatically pass function arguments based on their type (Odersky *et al.*, 2017). This makes implicits a perfect fit for the Effekt library.

To implicitly look up capabilities, for every effect signature, we define functions like:

```
def Amb given (a: Amb): a.type = a
```

Calling the nullary function `Amb` implicitly searches for a value of the equally named type in the current scope and returns it. As before, the return type is a singleton type to allow the necessary unification of path-dependent types our effect system relies on. Using these helpers, we can now rewrite the above example to:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
// AB ::= a AB | b
def parseAB given Amb given Exc given Input = alternative(
  accept('a') andThen parseAB map { _ + 1 },
  accept('b') map { x ⇒ 0 })

def accept(exp: Char) given Input given Exc = Input.read() flatMap { t ⇒
  if (t == exp) pure(()) else Exc.raise("Expected " + exp) }

def alternative[A, E](fst: A / E, snd: A / E) given Amb =
  Amb.flip() flatMap { b ⇒ if (b) fst else snd }
```

Note how all the arguments to the recursive call to `parseAB` (and to the functions `accept` and `alternative`) are now provided implicitly. The signature of `accept` informs us that it relies on an instance of `Input` and an instance of `Exc` being in scope at the call site.[4] At the same time, it brings these two instances in scope for the method body, so `Input.read` and `Exc.raise` will resolve to method calls on the corresponding implicit argument. Here, `Input` is the nullary method call to a boilerplate function as defined above. For the purpose of this paper, it is enough to understand that implicit search is performed at compile time, is lexically scoped, and type-directed. Implicit resolution results in a program like the explicit capability-passing variant above. We can also choose to bind implicit parameters to explicit names. In fact, this is necessary for the method `parseAB` which is recursive and thus requires a type annotation:

```
def parseAB given (a: Amb) given (e: Exc) given (i: Input)
  : Int / (a.effect & e.effect & i.effect)
```

Binding capabilities to names also enables us to fall back to passing them explicitly (e.g., `accept('a') given a given e`). This is important to resolve conflicts in case of ambiguous implicits, which would otherwise result in a compile time error.

6.3.2 Reducing the overhead by composition

Another strategy to reduce the burden of passing capabilities is by composition. We can define a trait that contains the necessary capabilities as members:

```
trait Parser { val amb: Amb; val exc: Exc; val input: Input }
```

Now all three methods can be refactored to only take one (potentially implicit) argument of type `Parser`, manually projecting the fields where necessary. Interestingly, the type signature of `accept` then becomes

```
def accept(exp: Char)(p: Parser): Unit / (p.exc.effect & p.amb.effect)
```

This illustrates the flexibility of path-dependent types in Scala. The stable path can have an arbitrary length.

[4] Since Scala 3, naming implicitly bound variables is optional. The signature thus roughly corresponds to `def accept(exp: Char)(implicit $1: Input, $2: Exc)` in Scala 2.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

6.4 Effect handlers and object orientation

Effekt is an embedding of effect handlers in a language with support for object-oriented programming. Naturally, the question arises how these two features interact.

Object-oriented programming has a strong focus on encapsulation. In particular, the concrete implementation of an object and its internal state is often hidden behind an interface. That is, the implementation can differ with the granularity of a single object. Another important feature is that objects are first class and typically are stored on the heap. In contrast, effects and handlers are tied to a stack discipline. Effect handler implementations can capture parts of the stack as a continuation, handlers delimit segments of the stack and effect typing asserts that these stack operations are safe, which effects are used by an object's implementation can either be seen as part of the public interface or as a private implementation detail. It is a design decision the programmer should make. However, if the effects used by an object are hidden behind an interface, how can we assert effect safety? For instance, if an object closes over a capability, the object's lifetime needs to be restricted to the capability's lifetime. Otherwise, the use of the capability might not be effect-safe.

The following interface will serve as a running example to discuss possible design choices when safely combining effect handlers with object-oriented programming:

```
trait Person { def greet(other: String): Unit }
```

6.4.1 Alternative 1: Effects as part of the public interface

An implementation might want to use the following effect to print the greeting.

```
trait Console extends Eff { def print(msg: String): Unit / effect }
```

However, the method `greet` as declared above does not mention the `Console` effect. Of course, we can change the interface accordingly.

```
trait Person {
  def greet(other: String)( out: Console ): Unit / out.effect
}
```

Now, the `Console` effect is part of the public interface and all implementations of `Person` can make use of it to implement the method `greet`. The effect has to be handled by the caller of `greet`. In this variant, it is possible to have multiple different implementations of `Person` and store the instances in data structures on the heap.

6.4.2 Alternative 2: Hiding effects behind an interface

Changing the interface of `Person` to mention the effects used by a particular implementation leaks implementation details. This problem also occurs with checked exceptions in Java. We can think of `Console` as a checked exception that is not mentioned in the interface of `greet`. Java programmers often resort to wrapping checked exceptions in unchecked ones to work around this problem (Zhang *et al.*, 2016). The exceptions thrown by an implementation can be considered an implementation detail that we might want to encapsulate. In Effekt, we can hide the effects behind an abstract type member `effect`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Person {
  type effect
  def greet(other: String): Unit / effect
}
```

An implementation of `Person` closes over the effect capabilities, just like a handler does:

```
trait MyPerson extends Person {
  val out: Console
  type effect = out.effect
  def greet(other: String) = out.print("Hello " + other)
}
```

Assuming a handler for the `Console` effect, in the following example, the lifetime of an instance of `MyPerson` is now coupled to the lifetime of the capability `o`.

```
withConsole { o ⇒ ... val p = new MyPerson { val out: o.type = o } ... }
```

For instance, the object `p` must not be used outside the scope of `withConsole`, which is ensured by our effect system: `out.effect` is an abstract type that only unifies with this one particular `o` created at the call to `withConsole`.

To be able to eventually handle the effects used by the implementation, users always need to have *stable paths* to an object.

```
def user(p1: Person, p2: Person): Unit / (p1.effect & p2.effect) =
  p1.greet("Alice") andThen p2.greet("Bob")
```

In this example p_1 and p_2 are arguments of method `user` and have stable paths that can be referred to in the return type. In general, path stability excludes objects from being stored in mutable references or containers like lists. While we can store p_1 in a mutable variable, the effect system will prevent us from calling any effectful methods on it.

6.4.3 *Alternative 3: Grouping objects by their effect implementations*

Alternative 1 requires all objects to use the same effects in their implementation and Alternative 2 allows each object to individually differ in their effect implementation. Both solutions also have drawbacks: the former constrains the implementer while the latter imposes restrictions on the user. As a compromise between the two, we can generalize over the effect implementation and thereby group objects by their effect implementations.

```
trait Person[E] { def greet(other: String): Unit / E }
```

Like with abstract type members, implementing classes can instantiate `E` to the desired implementation effects. Like with the first alternative, objects of type `Person[Console]` leak the implementation detail that they use the `Console` effect in their implementation.

Programs using instances of `Person` can be polymorphic in the effect type:

```
def user[E](p1: Person[E], p2: Person[E]): Unit / E =
  p1.greet("Alice") andThen p2.greet("Bob")
```

While we now can store objects of type `Person[E]` in lists of type `List[Person[E]]`, this requires all instances to have the same effect implementation. Furthermore, instances of

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

`Person[o.effect]` are still coupled to the lifetime of the capability `o`. An implementation that depends on more than one effect can only be used in the intersection of the respective handler regions.

7 Related work

In this section, we discuss closely related work. In particular, we compare our approach of capability passing to other implementations of effect handlers and relate our effect handlers design for object-oriented languages to others.

Shallow Embedding of Effect Operations. Many implementations of libraries and languages for effect handlers are based on a deep embedding of effect operations. In contrast, as pointed out in Section 6.2.2, by performing capability passing, Effekt builds on a shallow embedding of effect operations. Similarly, Kammar *et al.* (2013) base their library implementation of algebraic effect handlers on Haskell's type classes, effectively performing a shallow embedding. Using type classes and the associated dictionary passing helps Kammar *et al.* (2013) to achieve good performance results, since it prevents the materialization of constructors for effect operations. It also avoids any search for the matching handler implementation in some kind of handler stack, as it is done in Koka (Leijen, 2017a, c).

Continuations and Delimiters. Our implementation of delimited control is based on Dybvig *et al.* (2007), who in turn present a monadic implementation of a variant of multi-prompt delimited control by Gunter *et al.* (1995). Dybvig *et al.* (2007) present a very general framework that can express many different control operators. In particular, capturing the continuation with their control operator `withSubCont` removes both the corresponding outer delimiter and excludes it from the captured continuation. In contrast, our effect operation `switch` captures and removes the continuation up to *and including* the corresponding delimiter `handle`. It is thus operationally closer to shift_0 by Danvy & Filinski (1989, Appendix C) and spawn by Hieb & Dybvig (1990) as illustrated by the following equation:

```
def spawn(body) = handle { s ⇒ body(s.switch) }
```

As highlighted by Kammar *et al.* (2013), shift_0 matches the semantics of *deep handlers* where the same effect is already handled in the continuation. Materzok & Biernacki (2011) present an effect system for shift_0 that supports answer-type modification, while our effect-safe version of `Control` does not. In turn, Effekt allows handlers to switch to a specific scope, while shift_0 always captures the continuation up to the closest delimiter.

Effect-Safe Multi-Prompt Delimited Control. Like our presentation in Section 3, Dybvig *et al.* (2007) guarantee answer-type safety by indexing prompts with the expected answer type. Furthermore, they use rank-2 types to prevent prompts from being used across different instances of `run`. But, as they observe, this is not enough to achieve effect safety, which they explicitly leave to future work. We generalize the idea of region safety and guarantee that capabilities cannot be used outside of the scope that they are

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

created in. Instead of rank-2 types, we use abstract type members. This lets us easily nest scopes using intersection types. In general, Scala has better support for path-dependent function types than for rank-2 types. For instance, we can use the lambda syntax (e.g., `amb ⇒ amb.flip()`), which is not possible with rank-2 types that need to be instantiated as anonymous inner classes. Compared to path-dependent function types, the use of rank-2 types is thus more verbose and hinders type inference.

Region Safety for Effects. Many languages with effect handlers base their effect system on some form of row polymorphism. Prominent examples are Koka (Leijen, 2014), Frank (Lindley *et al.*, 2017), and Links (Hillerström & Lindley, 2016). In contrast, effect-safe library embeddings like Extensible Effects (Kiselyov *et al.*, 2013; Kiselyov & Ishii, 2015) use various forms of open union types to track the list of unhandled effects. In Effekt, we index effectful computations with an intersection of all capabilities used by a computation. This way, capabilities cannot be used outside of their handler *region*.

Region Safety for Resources. To achieve resource safety, Kiselyov & Shan (2008) generalize from a single region (Moggi & Sabry, 2001) to multiple nested regions. They achieve region polymorphism and region subtyping together with good type inference for their library in Haskell. On the type level, they represent nested regions as nesting of a monad transformer while we represent nested delimiters by an intersection of abstract type members. To achieve region polymorphism, they reuse Haskell's polymorphism and to achieve region subtyping they use Haskell's type class instance search. In contrast, to achieve effect polymorphism, we reuse Scala's polymorphism and to achieve subtyping, we reuse subtyping for intersection types built into Scala.

Region Safety for Variable Scopes. Parreaux *et al.* (2017) apply a strategy very similar to ours to guarantee scope safety in type-safe meta-programming. The type parameter `Ctx` of their type `Code[+Typ, -Ctx]` is used to track the set of free variables:

```
class Variable[A] {
  type Ctx;
  def substitute[T,C](pgrm: Code[T, Ctx & C], v: Code[A, C]): Code[T,C]
}
```

As can be seen from the type of `substitute`, substitution of free variables removes `Ctx` from the intersection type and thus corresponds to handling of effects in Effekt.

Capability Passing. Like in this paper, Osvald *et al.* (2016) perform capability passing in Scala: a capability serves as a constructive proof that the holder is entitled to use the actions associated with the capability. To prevent leaking of capabilities, Osvald *et al.* (2016) introduce a type-based escape analysis as an alternative approach to traditional effect systems: arguments to functions can be marked as second class (or "local"). The type checker then guarantees that the capability cannot leave the dynamic scope of the function call. Liu (2016) presents a different approach to capability-based effect safety, by distinguishing between functions that can capture capabilities and others that cannot (called "stoic"). In our design of Effekt, we adopt the capability passing of Osvald *et al.* (2016). Second-class

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

values might be an interesting alternative to the effect system presented in this paper. However, they are not available in Scala, while our library can readily be used.

Abstraction-Safe Effect Handlers via Tunneling. The dynamic and static semantics of Effekt is closely related to $\lambda_{\Downarrow\Uparrow}$ presented by Zhang & Myers (2019). As in previous versions of Effekt (Brachthäuser & Schuster, 2017), capabilities in $\lambda_{\Downarrow\Uparrow}$ are pairs of a label and the handler implementation. Also like in Effekt, they are explicitly passed to the use site of the effect operation. Handling an effect introduces a fresh label. Like scope capabilities in Effekt, the label is used on the term level to delimit the scope of captured continuations. The label is used on the type level to track the set of unhandled effects, similar to the type member `effect` in the present paper. Due to the embedding of Effekt in Scala, scope capabilities are first class while labels in $\lambda_{\Downarrow\Uparrow}$ are not first class. Instead, the binding of a label by means of `try` and the use of a label in a handler implementation is statically scoped. For better comparison of $\lambda_{\Downarrow\Uparrow}$ and Effekt, Appendix A provides a more direct embedding into Scala.

Effect Handlers and Object-Oriented Programming. In earlier work on Effekt (2017; 2018), we started to explore the combination of effect handlers and object orientation. However, those versions of Effekt did not guarantee effect safety. The present paper shows how to add effect safety to Effekt and support effect polymorphism. Indexing the monad for delimited control with the set of used effects is essential to guarantee effect safety. It is not immediate to us how the effect system can be embedded in a direct style version of the library (Brachthäuser *et al.*, 2018) because there is no monadic type that could carry the set of used effects. As highlighted in Section 6, effect-safe programming with effect handlers in a language with objects comes with new challenges—mediating encapsulation and flexible use of objects. Inostroza & van der Storm (2018) also combine effect handlers and object orientation in the language JEff. In JEff, the continuation takes an updated copy of the effect handler as additional argument. Parametrizing continuations by the handler instances can model dynamically scoped state (Kiselyov *et al.*, 2006) and also allows handlers to change their implementation for the rest of the computation, similar to shallow handlers. The effect system of JEff does not feature effect polymorphism and hence problems with effect encapsulation and effect parametricity do not arise.

8 Conclusion

In this article, we presented Effekt, a monadic library for programming with effect handlers in Scala that features effect polymorphism, effect subtyping, and effect safety. We use intersection types and path-dependent types to track the set of effects a program might use. This allows us to directly reuse Scala's support for polymorphism for effect polymorphism and Scala's support for subtyping for effect subtyping. Combining effect handlers with object-oriented programming both offers new ways to modularize effectful programs but also comes with new challenges.

Conflicts of Interest

None

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

References

Amin, N. & Tate, R. (2016) Java and Scala's type systems are unsound: The existential crisis of null pointers. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. New York, NY, USA: ACM, pp. 838–848.

Bauer, A. & Pretnar, M. (2015) Programming with algebraic effects and handlers. *J. Log. Alg. Methods Program.* **84**(1), 108–123.

Biernacki, D., Piróg, M., Polesiuk, P., & Sieczkowski, F. (2017) Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM Program. Lang.* **2**(POPL), 8:1–8:30.

Biernacki, D., Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Abstracting algebraic effects. *Proc. ACM Program. Lang.* **3**(POPL), 6:1–6:28.

Brachthäuser, J. I. & Schuster, P. (2017) Effekt: Extensible algebraic effects in Scala (short paper). In Proceedings of the International Symposium on Scala. New York, NY, USA: ACM.

Brachthäuser, J. I., Schuster, P. & Ostermann, K. (2018) Effect handlers for the masses. *Proc. ACM Program. Lang.* **2**(OOPSLA), 111:1–111:27.

Carette, J., Kiselyov, O. & Shan, C.-C. (2007) Finally tagless, partially evaluated. In Proceedings of the Asian Symposium on Programming Languages and Systems. LNCS, vol. 4807. Berlin, Heidelberg: Springer, pp. 222–238.

Convent, L., Lindley, S., McBride, C. & McLaughlin, C. (2019) *Doo Bee Doo Bee Doo*. Technical report. The University of Edinburgh.

Danvy, O. & Filinski, A. (1992) Representing control: A study of the CPS transformation. *Math. Struct. Comput. Sci.* **2**(4), 361–391.

Danvy, O. & Filinski, A. (1989) A functional abstraction of typed contexts. Diku rapport 89/12, diku, University of Copenhagen.

Danvy, O. & Filinski, A. (1990) Abstracting control. In Proceedings of the Conference on LISP and Functional Programming. New York, NY, USA: ACM, pp. 151–160.

Dolan, S., Eliopoulos, S., Hillerström, D., Madhavapeddy, A., Sivaramakrishnan, K. C. & White, L. (2017) Concurrent system programming with effect handlers. In Proceedings of the Symposium on Trends in Functional Programming. LNCS, vol. 10788. Springer.

Dolan, S., White, L. & Madhavapeddy, A. (2014) Multicore OCaml. In OCaml Workshop.

Dolan, S., White, L., Sivaramakrishnan, K. C., Yallop, J. & Madhavapeddy, A. (2015) Effective concurrency through algebraic effects. In OCaml Workshop.

Dybvig, R. K., Peyton J., Simon L. & Sabry, A. (2007) A monadic framework for delimited continuations. *J. Funct. Program.* **17**(6), 687–730.

Felleisen, M. (1988) The theory and practice of first-class prompts. In Proceedings of the Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 180–190.

Forster, Y., Kammar, O., Lindley, S. & Pretnar, M. (2017) On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.* **1**(ICFP), 13:1–13:29.

Friedman, D. P., Haynes, C. T. & Kohlbecker, E. (1984) Programming with continuations. In *Program Transformation and Programming Environments*, Pepper, P. (ed), Berlin, Heidelberg: Springer-Verlag.

Gaster, B. R. & Jones, M. P. (1996) *A Polymorphic Type System for Extensible Records and Variants*. Technical report NOTTCS-TR-96-3.

Gunter, C. A., Rémy, D. & Riecke, J. G. (1995) A generalization of exceptions and control in ML-like languages. In Proceedings of the Conference on Functional Programming Languages and Computer Architecture. New York, NY, USA: ACM, pp. 12–23.

Haller, P. & Loiko, A. (2016) LaCasa: Lightweight affinity and object capabilities in Scala. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. New York, NY, USA: ACM, pp. 272–291.

Hieb, R. & Dybvig, R. K. (1990) Continuations and concurrency. In Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming. PPOPP'90. New York, NY, USA: ACM, pp. 128–136.

Hieb, R., Dybvig, R. K. & Anderson III, C. W. (1994) Subcontinuations. *Lisp Symb. Comput.* **7**(1), 83–110.

Hillerström, D. & Lindley, S. (2016) Liberating effects with rows and handlers. In Proceedings of the Workshop on Type-Driven Development. New York, NY, USA: ACM.

Hillerström, D., Lindley, S., Atkey, B. & Sivaramakrishnan, K. C. (2017) Continuation passing style for effect handlers. In Formal Structures for Computation and Deduction, LIPIcs, vol. 84. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Hudak, P. (1998) Modular domain specific languages and tools. In Proceedings of the Conference on Software Reuse. IEEE Computer Society, pp. 134–142.

Inostroza, P. & van der Storm, T. (2018). JEff: Objects for effect. In Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. Onward! 2018. New York, NY, USA: ACM.

Johnson, G. F. & Duggan, D. (1988). Stores and partial continuations as first-class objects in a language and its environment. In Proceedings of the Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 158–168.

Kammar, O., Lindley, S. & Oury, N. (2013) Handlers in action. In Proceedings of the International Conference on Functional Programming. New York, NY, USA: ACM, pp. 145–158.

Kiselyov, O. & Ishii, H. (2015) Freer monads, more extensible effects. In Proceedings of the Haskell Symposium. New York, NY, USA: ACM, pp. 94–105.

Kiselyov, O. & Shan, C.-c. (2008) Lightweight monadic regions. In Proceedings of the Haskell Symposium. Haskell'08. New York, NY, USA: ACM.

Kiselyov, O. & Sivaramakrishnan, K. C. (2016). Eff directly in OCaml. In ML Workshop.

Kiselyov, O. & Sivaramakrishnan, K. C. (2018) Eff directly in OCaml. In Proceedings of the ML Family Workshop/OCaml Users and Developers Workshops, Asai, K. & Shinwell, M. (eds). Electronic Proceedings in Theoretical Computer Science, vol. 285. Open Publishing Association, pp. 23–58.

Kiselyov, O., Sabry, A. & Swords, C. (2013) Extensible effects: An alternative to monad transformers. In Proceedings of the Haskell Symposium. New York, NY, USA: ACM, pp. 59–70.

Kiselyov, O., Shan, C.-c. & Sabry, A. (2006) Delimited dynamic binding. In Proceedings of the International Conference on Functional Programming. New York, NY, USA: ACM, pp. 26–37.

Kobori, I., Kameyama, Y. & Kiselyov, O. (2016) Answer-type modification without tears: Prompt-passing style translation for typed delimited-control operators. arxiv preprint arxiv:1606.06379.

Launchbury, J. & Sabry, A. (1997) Monadic state: Axiomatization and type safety. In Proceedings of the International Conference on Functional Programming. ICFP'97. New York, NY, USA: ACM, pp. 227–238.

Leijen, D. (2014) Koka: Programming with row polymorphic effect types. In Proceedings of the Workshop on Mathematically Structured Functional Programming.

Leijen, D. (2016) *Algebraic Effects for Functional Programming*. Technical reportt MSR-TR-2016-29. Microsoft Research technical report.

Leijen, D. (2017a) Implementing algebraic effects in C. In Proceedings of the Asian Symposium on Programming Languages and Systems. Cham, Switzerland: Springer International Publishing, pp. 339–363.

Leijen, D. (2017b). Structured asynchrony with algebraic effects. In Proceedings of the Workshop on Type-Driven Development. New York, NY, USA: ACM, pp. 16–29.

Leijen, D. (2017c). Type directed compilation of row-typed algebraic effects. In Proceedings of the Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 486–499.

Leijen, D. (2018). First class dynamic effect handlers: Or, polymorphic heaps with dynamic effect handlers. In Proceedings of the Workshop on Type-Driven Development. New York, NY, USA: ACM, pp. 51–64.

Liang, S., Hudak, P. & Jones, M. (1995)Monad transformers and modular interpreters. In Proceedings of the Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 333–343.

Lindley, S. (2018) Encapsulating effects. *Dagstuhl Reports*, **8**(4).

Lindley, S., McBride, C. & McLaughlin, C. (2017) Do be do be do. In Proceedings of the Symposium on Principles of Programming Languages. New York, NY, USA: ACM, pp. 500–514.

Liu, F. (2016) *A Study of Capability-Based Effect Systems*. M.Phil. thesis, École Polytechnique Fédérale de Lausanne, Switzerland.

Materzok, M. & Biernacki, D. (2011) Subtyping delimited continuations. In Proceedings of the International Conference on Functional Programming. New York, NY, USA: ACM, pp. 81–93.

Moggi, E. & Sabry, A. (2001) Monadic encapsulation of effects: A revised approach (extended version). *J. Funct. Program.* **11**(6), 591–627.

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

Odersky, M. & Zenger, M. (2005a) Independently extensible solutions to the expression problem. In Proceedings of the Workshop on Foundations of Object-Oriented Languages.

Odersky, M. & Zenger, M. (2005b) Scalable component abstractions. In Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications. New York, NY, USA: ACM, pp. 41–57.

Oliveira, B. C. d. S., & Cook, W. R. (2012) Extensibility for the masses: Practical extensibility with object algebras. In Proceedings of the European Conference on Object-Oriented Programming. LNCS, vol. 7313. Springer, pp. 2–27.

Osvald, L., Essertel, G., Wu, X., Alayón, L. I. G.& Rompf, T. (2016) 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. New York, NY, USA: ACM, pp. 234–251.

Parreaux, L., Voizard, A., Shaikhha, A. & Koch, C. E. (2017). Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* **2**(POPL), 13:1–13:33.

Piróg, M., Polesiuk, P. & Sieczkowski, F. (2019) Typed equivalence of effect handlers and delimited control. In Formal Structures for Computation and Deduction. LIPIcs. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 30:1–30:16.

Plotkin, G. & Power, J. (2003) Algebraic operations and generic effects. *Appl. Categor. Struct.* **11**(1), 69–94.

Plotkin, G. & Pretnar, M. (2009) Handlers of algebraic effects. In European Symposium on Programming. Springer-Verlag, pp. 80–94.

Plotkin, G. D. & Pretnar, M. (2013) Handling algebraic effects. *Log. Methods Comput. Sci.* **9**(4), 1–36.

Rapoport, M. & Lhoták, O. (2019). A path to DOT: formalizing fully-path-dependent types. Corr, abs/1904.07298.

Sitaram, D. & Felleisen, M. (1990) Control delimiters and their hierarchies. *Lisp Symb. Comput.* **3**(1), 67–99.

Wadler, P. (1998). *The expression problem*. Note to Java Genericity mailing list.

Wright, A. K. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inf. Comput.* **115**(1), 38–94.

Wu, N., Schrijvers, T. & Hinze, R. (2014) Effect handlers in scope. In Proceedings of the Haskell Symposium. Haskell'14. New York, NY, USA: ACM, pp. 1–12.

Zhang, Y. & Myers, A. C. (2019) Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* **3**(POPL), 5:1–5:29.

Zhang, Y., Salvaneschi, G., Beightol, Q., Liskov, B. & Myers, A. C. (2016) Accepting blame for safe tunneled exceptions. Proceedings of the Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, pp. 281–295.

```
type ℓ[T, E] = Scope[T, E]

trait 𝔽[-A, +B] { type effect; def apply(arg: A): Control[B, effect] }

case class Handler[-A, +B, T, E](
  label: ℓ[T, E],
  handler: A ⇒ (B ⇒ Control[T, E]) ⇒ Control[T, E]
) extends 𝔽[A, B] {
  type effect = label.effect
  def apply(x: A) = label.switch { k ⇒ handler(x)(k) }
}

def ⇩[R, E](prog: (l: ℓ[R, E]) ⇒ Control[R, l.effect & E]): Control[R, E] =
  handle { prog }
def ⇧[A, B](f: 𝔽[A, B]): A ⇒ Control[B, f.effect] = x ⇒ f.apply(x)
```

Fig. 8. Using `Control` to embed the $\lambda_{⇩⇧}$ calculus into Scala.

Appendix

Embedding the Tunneling Calculus

The effect system as presented in this paper is heavily influenced by the one of $\lambda_{⇩⇧}$ (Zhang & Myers, 2019). To ensure effect safety, Zhang & Myers use a simple form of dependent types: Using an effect handler `h` introduces `h.lbl` in the effect type which is effectively a set of labels. The dependent label corresponds to the abstract type member `h.effect` in Effekt. Like in Effekt, this dependent effect type can only be discharged by the very same delimiter (denoted by $⇩^{\ell}\, t$) that introduced the label. Zhang & Myers formalize $\lambda_{⇩⇧}$ and formally show effect parametricity. However, they do not provide an implementation of their calculus. We use intersection types and path dependent types to encode the ideas of the $\lambda_{⇩⇧}$ effect system and thereby make Effekt effect safe.

To facilitate comparison of $\lambda_{⇩⇧}$ with Effekt, Figure 8 embeds the $\lambda_{⇩⇧}$ calculus (Zhang & Myers, 2019) in Scala. We immediately use the `Control` monad and the `Scope` effect to express the operational semantics. We just present the practical embedding into Scala and leave a formal translation and corresponding soundness proofs to future work. However, assuming a sound subset of Scala that corresponds to $\lambda_{⇩⇧}$, we conjecture that our embedding faithfully models the calculus by Zhang & Myers. In particular, the restriction to a subset of Scala excludes the use of mutable state, recursive function definitions, recursive data-types, and exceptions. Also effect signatures have to be declared on the top level and should not use mixin composition, type members, subtyping, or any other advanced Scala feature.

Example

To ease comparison with the original calculus, we use ℓ as the type of labels and $\mathbb{F}$ as the type of effect signatures. Effect signatures `𝔽[A, B]` only declare a single-effect operation with argument type `A` and return type `B`. For example, we can express the signatures of the ambiguity and exception effect as:

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press

```
trait Amb extends 𝔽[Unit, Boolean]
trait Exc extends 𝔽[String, Nothing]
```

Handlers, modeled by the case class `Handler`, are pairs of labels and effect implementations. In this style, the handler for expressions can be expressed as:

```
def maybe[R, E](prog: (exc: Exc) ⇒ Control[R, exc.effect & E]) = ⇩ { l ⇒
  val exc = new Handler(l, msg ⇒ k ⇒ pure(None)) with Exc
  prog(exc) map { r ⇒ Some(r) }
}
```

Calling an effect operation amounts to calling ⇧ on the handler:

```
maybe { h ⇒ ... ⇧(h)("Failed!") ... }
```

While in $\lambda_{⇩⇧}$, ⇧ performs the continuation capture, in the embedding we perform the capturing in the implementation of `Handler.apply` by means of `label.switch`. This is necessary to have the available answer types (`T` and `E` in scope). We use subtyping `Handler <: 𝔽` to existentially hide the answer types and other implementation details of `Handler` when passing capabilities of type `𝔽`.

Refinement to Singleton Types. In the presentation of Effekt in this paper, we always performed explicit refinement to singleton types to establish type equalities of type members. Similarly, to prevent widening of the label singleton type to ℓ, the actual signature of `Handler` is:

```
case class Handler[-A, +B, T, E, L <: ℓ[T, E]](label: L, ...)‘
```

Additionally, at construction site, the type parameter `L` has to be explicitly provided as singleton type `l.type`:

```
val h = Handler[A, B, T, E, l.type](l, ???).effect
```

This establishes the type equality between `l.effect` and `h.effect`.

https://doi.org/10.1017/S0956796820000027 Published online by Cambridge University Press