Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism

JONATHAN IMMANUEL BRACHTHÄUSER, EPFL, Switzerland
PHILIPP SCHUSTER and KLAUS OSTERMANN, University of Tübingen, Germany

Effect handlers have recently gained popularity amongst programming language researchers. Existing type- and effect systems for effect handlers are often complicated and potentially hinder a wide-spread adoption. We present the language Effekt with the goal to close the gap between research languages with effect handlers and languages for working programmers. The design of Effekt revolves around a different view of effects and effect types. Traditionally, effect types express which *side effects* a computation might have. In Effekt, effect types express which *capabilities* a computation requires from its context. While this new point in the design space of effect systems impedes reasoning about purity, we demonstrate that it simplifies the treatment of effect polymorphism and the related issues of effect parametricity and effect encapsulation. To guarantee effect safety, we separate functions from values and treat *all* functions as second-class. We define the semantics of Effekt as a translation to System Ξ, a calculus in explicit capability-passing style.

CCS Concepts: • **Software and its engineering → Control structures**; **Abstraction, modeling and modularity**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: effect handlers, algebraic effects, effect polymorphism

ACM Reference Format:
Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 126 (November 2020), 30 pages. https://doi.org/10.1145/3428194

1 INTRODUCTION

Effects and handlers [Plotkin and Pretnar 2013] have become quite popular in programming language research, recently. Effectful expressions are expressions that *depend on* or *modify* the context they are evaluated in [Wright and Felleisen 1994]. Effect systems extend the guarantees of programming languages from type safety to effect safety: all effects (*e.g.*, exceptions) are eventually handled, and they are not accidentally handled by the wrong handler. Effect handlers allow novel ways to structure effectful programs as reusable libraries [Plotkin and Pretnar 2013], while guaranteeing effect safety and enabling equational reasoning with effect parametricity [Zhang and Myers 2019].

A well-known downside of current languages with effect handlers is that, arguably, their semantics (both dynamic and static) is too complicated to be put into practical use at scale. To guarantee effect safety, existing implementations incorporate sophisticated type- and effect systems. To master these effect systems, programmers need to study the corresponding languages intensively. The combination of effects with higher-order functions requires support for *effect polymorphism* – a delicate feature, which is particularly difficult to understand and reason about, as witnessed by

Authors' addresses: Jonathan Immanuel Brachthäuser, EPFL, IC LAMP1, INR 319, Station 14, 1015 Lausanne, Switzerland, jonathan.brachthauser@epfl.ch; Philipp Schuster, philipp.schuster@uni-tuebingen.de; Klaus Ostermann, klaus.ostermann@uni-tuebingen.de, University of Tübingen, Sand 13, 72076 Tübingen, Germany.

2475-1421/2020/11-ART126
https://doi.org/10.1145/3428194

recent publications [Biernacki et al. 2020; Brachthäuser et al. 2020a; Zhang and Myers 2019]. Some languages (Frank [Lindley et al. 2017], Koka [Leijen 2017b], and Helium [Biernacki et al. 2020]) attempt to hide effect polymorphism behind syntactic sugar. However, this masquerade often breaks down in more sophisticated use cases and details of effect polymorphism leak to the startled user.

In this paper, we take a radically different view on effect systems for languages with effect handlers. Traditionally, effect types express which *side effects* a computation might have, besides computing the resulting value. Instead, we take on a different interpretation where effect types express which *capabilities* a computation requires from its context. This different perspective opens up a novel design space, offering different trade-offs, which this paper starts to explore. We present the design of Effekt[1], a new language with support for effects and handlers. While Effekt has the established dynamic semantics of lexically scoped effects [Brachthäuser and Schuster 2017], it is equipped with an effect system, which we claim is significantly more lightweight than those of existing languages with effect handlers. Effekt avoids the complexity that comes with *parametric* effect polymorphism simply by omitting the feature from the language.

1.1 Effects as Requirements: The Contextual Reading

In languages with effect systems, effectful functions generally have a type like $\alpha \rightarrow \beta / \epsilon$, where α and β are meta variables representing value types, and ϵ is a meta variable that represents a collection of effects. We identify two possible readings of such a type signature.

The Traditional Reading. Traditionally, the signature would be read as follows:

> "Given a value of type α, the function produces a value of type β and has effects ϵ".

Effects are often seen as a *side effect* or additional "output" of a function: effectful expressions *modify the evaluation context*. In particular, an empty ϵ implies that the function cannot have effects – it is *pure*. In contrast, a non-empty ϵ suggests that a function is *effectful*.

The Contextual Reading. In this paper, we propose a novel reading of the types of effectful functions given above, inspired by Osvald et al. [2016].

> "Given a value of type α, the function produces a value of type β and requires the calling context to handle effects ϵ".

We interpret effects as a *requirement* to the caller or additional "input" to a function: effectful expressions *depend on the evaluation context*. In particular, an empty ϵ implies that the function does not impose any requirements on its caller – it is *contextually pure*. Contextual purity does not imply purity; a contextually pure function may be side-effecting, but those effects are handled *elsewhere*. In contrast, a non-empty ϵ suggests that the function is contextually effectful, meaning that the context is responsible for handling the effects.

1.2 An Example

The following example program in Effekt uses a higher-order function `eachLine`, which takes two arguments: a file `someFile` and a *block* (in braces) and calls the block on each line in the file. In the example, we cannot process empty lines and therefore abort by throwing an exception.

```
try {
  eachLine(someFile) { (line) ⇒ if (line == "") { do Exception() } else { ... } }
} with Exception { () ⇒ println("empty line") }
```

What should the signature of `eachLine` be and how should that type be interpreted?

[1] Our language should not be confused with an equally named Scala library [Brachthäuser and Schuster 2017; Brachthäuser et al. 2020a], which also performs capability passing, but is based on the traditional notion of effect polymorphism.

The Traditional Reading. In an effect language that uses the traditional reading (we use Koka [Leijen 2017b] here for illustration), a conceivable signature would be this:

```
eachLine : (file, string → <> ()) → <> ()
```

The type of the function parameter has an empty effect row <>, which means it cannot have any effects. Under this type signature our example would not type check, since we pass an anonymous function (block) that throws exceptions and thus *does* have an effect. We could change the signature of `eachLine` to indicate that the function parameter may throw exceptions. Since `eachLine` calls its function parameter, we have to adapt the resulting effect type as well:

```
eachLine : (file, string → <exception> ()) → <exception> ()
```

Specializing signatures of higher-order functions to every use-site is practically not feasible and so Koka offers support for *parametric effect polymorphism*.

```
eachLine : forall<e> (file, string → e ()) → e ()
```

The Contextual Reading. Clearly, demanding the parameter of `eachLine` to be pure is too strict, as the example illustrates. Extending the signature for a particular use case is not a modular solution. Effect polymorphism comes with its own set of problems, which we discuss in a moment. In any case, the traditional reading does not allow to express that the parameter of `eachLine` should be *contextually pure*, which is expressed as follows in Effekt:

```
def eachLine(file: File) { f: String ⇒ Unit / {} }: Unit / {}
```

Effects provided by `eachLine` — Effects required by `eachLine`

where `{}` denotes the empty set of effects and types of block parameters (like `f`) are enclosed in braces[2]. Applying the contextual reading, the return type of `eachLine` communicates that it does not require any effects. In consequence, we can call `eachLine` in any context. More interestingly, the type of the block parameter `f` also mentions no effects. The function `f` is contextually pure: While the caller of `f` might *observe* additional effects (for instance through mutable state), it cannot *handle* them. Instead, all effects that are used by `f` have to be handled at the call-site of `eachLine`. In our example, the `Exception` effect used to signal empty lines is handled at the call site.

1.3 Parametric vs Contextual Effect Polymorphism

Effect polymorphism means that programmers can reuse `eachLine` with different function arguments with different effects. In languages like Koka, with support for *parametric* effect polymorphism, this amounts to instantiating the effect variable `e` to the effects of choice (*i.e.*, `exception` in our example). With parametric effect polymorphism, signatures of functions make it *explicit*, which effects they do not care about, which might seem counterintuitive. Parametric effect polymorphism complicates function signatures and care must be taken to avoid accidental capture [Zhang and Myers 2019] and guarantee encapsulation [Convent et al. 2020]. We agree with Lindley et al. [2017], who state that users should not be confronted with the details of effect polymorphism

> In designing Frank we have sought to maintain the benefits of effect polymorphism whilst avoiding the need to write effect variables in source code.

and with Leijen [2017b], who states that

> In practice though we wish to simplify the types more and leave out "obvious" polymorphism.

[2]Enclosing the type of block parameters in braces mimics the syntax at the call site and furthermore helps to distinguish blocks from traditional first-class functions / lambdas.

Languages like Koka and Frank, attempt to hide the details of parametric effect polymorphism behind syntactic sugar. However, once programs get sufficiently complicated, the syntactic abstraction breaks down and effect polymorphism becomes visible to the user (for instance in error messages). To the best of our knowledge, all languages with support for effects and handlers and static effect-typing support this *parametric* form of effect polymorphism.

Contextual Effect Polymorphism. In contrast, Effekt offers *contextual effect polymorphism*. Effekt does not have language constructs for type-level effect variables or quantification over effects. Still, it supports effect polymorphic reuse of functions and guarantees effect safety. We say that effect polymorphism in Effekt is *contextual*. Users never have to deal with parametric effect polymorphism as the feature simply does not exist: polymorphism implicitly arises from the calling context, as we are going to elaborate in the formal part of the paper. Our running example type- and effect-checks without modification in Effekt.

1.4 Lexically Scoped Handlers through Explicit Capability Passing

To establish a dynamic semantics consistent with contextual effect polymorphism, Effekt programs are translated to a calculus System Ξ (Section 4) in explicit capability-passing style [Brachthäuser and Schuster 2017; Brachthäuser et al. 2020a], implementing lexically scoped handlers [Biernacki et al. 2020; Zhang and Myers 2019]. Translating our running example to System Ξ shows that handler implementations are represented as *blocks* and passed to the call-site of the effect operation:

```
handle { Exception ⇒
  eachLine(someFile, { line ⇒ if (line ≡ "") Exception() else ... })
} with { (resume) ⇒ println("empty line") }
```

Here, we assume the above signature of eachLine: the Exception effect is *not* handled by the call to eachLine. Instead, the handler for Exception creates a *capability* and binds it to the equally named term variable, highlighted in gray. Explicit capability passing is essential to establish a *lexical* connection between an effect handler and the operations it handles. The block passed to eachLine simply *closes* over the capabilities that are in scope at its definition site. Closing over capabilities avoids accidental handling, a problem that arises when handlers are allocated on the stack and dynamically searched at runtime. Capability passing also enables *effect parametric reasoning*: from inspecting the type of eachLine, we know that there is no way it can modify the semantics of the Exception effect [Biernacki et al. 2020; Brachthäuser et al. 2020a; Zhang and Myers 2019].

The semantics of Effekt is designed to be consistent with the expectations implied by contextual reading of effect annotations. In particular, changing the signature of eachLine to

```
def eachLine(file: File) { f: String ⇒ Unit / { Exception } }: Unit / {}
```

signals that eachLine now can handle Exception effects. This is also reflected in the translation:

```
handle { Exception ⇒
  eachLine(someFile, { (line, Exception) ⇒ if (line ≡ "") Exception() else ... })
} with { (resume) ⇒ println("empty line") }
```

The block now binds an Exception capability, shadowing the one of the surrounding handler.

1.5 Purity: Traditional vs. Contextual Reading

While a function with an empty set of effects is pure in the traditional reading, the same is not true for the contextual reading. A contextually pure function can still have effects, it merely places no requirements on its caller. Purity only holds relative to the function's definition site.

```
def hof1 { f: () ⇒ Unit } = { f(); 2 }     def hof2 { f: () ⇒ Unit } = { 2 }
```

In the traditional reading, $\texttt{hof}_1$ and $\texttt{hof}_2$ would be equivalent since `f` must be pure. However, in our language, the following code demonstrates that they are not equivalent in the contextual reading.

```
try { hof1 { do Exception() } } with Exception { () ⇒ 1 }
```

This code will yield 1 but if we replace the invocation of $\texttt{hof}_1$ by $\texttt{hof}_2$ it will yield 2.

1.6 Contextual Reading and Second-Class Functions

To support contextual effect polymorphism and at the same time guarantee effect safety, the type- and effect system of Effekt strictly separates values from blocks (Section 3.2). Furthermore, blocks are considered second-class [Osvald et al. 2016] and can neither be returned nor stored in data structures. The following example illustrates, why first-class blocks would be problematic:

```
try { def leak(): Unit / {} = { do Exception() }; leak } with Exception { ... };
```

Here, we define the block `leak`, which lists no effects since they are handled in its definition context. However, using the leaked block outside of the corresponding `try` is not safe and leads to a runtime error. To prevent this, while Effekt supports higher-order functions (like `eachLine`), it does not support first-class functions. The loss of expressivity is the price for an effect system, which we argue is significantly more lightweight, while offering full safety guarantees and reasoning with effect parametricity. While we can imagine to recover the lost expressivity and separately add first-class functions [Osvald et al. 2016] or infer whether functions are first- or second-class [Zhang et al. 2016], in this paper we explicitly refrain from doing so, laying ground for future explorations.

1.7 Overview and Contributions

The remainder of this paper is structured as follows. Section 2 offers an example-driven introduction to programming with effect handlers in the Effekt language. It illustrates how our reading of effects goes hand-in-hand with our translation to explicit capability-passing style. We then formally define Effekt (Section 3) and provide the operational semantics of Effekt by translating programs to System Ξ (Section 4) – a calculus in explicit capability-passing style. We prove soundness of System Ξ and of our translation from Effekt to System Ξ (Section 5). We practically evaluate Effekt by presenting a practical implementation scaling Effekt to a fully fledged language (Section 6). Finally, in Section 7 we discuss implications of the contextual reading and compare Effekt to related work on effect handlers. This paper makes the following contributions:

- A formal presentation of the language Effekt with effect handlers and effect safety, but without *parametric* effect polymorphism (Section 3).
- An *algorithmic* effect system, which is rooted in a *different understanding* of effects, reading effects as requirements that need to be fulfilled by the context. Effect types in Effekt are *sets* not rows or lists, which we argue makes them easier to understand (Section 3.2).
- A calculus System Ξ in *explicit capability-passing style*, supplying handler implementations as additional arguments to effectful functions (Section 4). The type system of System Ξ does not include effects or effect types. Instead, handler implementations are represented as blocks and effect safety is established by treating blocks as second class (Section 4.2).
- A definition of the semantics of Effekt by translation into System Ξ (Section 5), which is the first *formalization* of a translation from a language with effect handlers to *explicit* capability-passing style (Section 5).
- A mechanized proof of soundness of System Ξ, a proof of well-typedness preservation of the translation, and a proof of effect safety of Effekt (Sections 4 and 5).

- An implementation of Effekt, compiling programs to JavaScript and using a library implementation of multi-prompt delimited control (Section 6).
- A practical evaluation by extending the calculus to a fully fledged language and implementing several medium-sized case studies (Section 6.3).

Due to lack of space, some definitions and proofs have been omitted. A full formalization as well as additional proofs can be found in an extended technical report [Brachthäuser et al. 2020b].

2 PROGRAMMING WITH EFFECTS AND HANDLERS IN Effekt

In Section 1, we have seen how exceptions are raised and handled in Effekt. However, exceptions are just an instance of the more powerful concept of *effects and handlers* [Plotkin and Pretnar 2013]. In general, with effect handlers programs are structured into three components: *Effect signatures* that define available *effect operations*, *effectful programs* that use effect operations, and *effect handlers* that give meaning to the effect operations. In this section, we introduce programming with effect handlers in Effekt and illustrate the contextual reading of effects in several examples. The examples illustrate that, despite the contextual reading, programming with effects and handlers in Effekt is not much different from existing languages. Differences only become visible in examples that require effect polymorphism, like higher-order functions. As a running example, we adopt the approach by Leijen [2016] and implement a parser combinator library using effect handlers. Our goal is to parse a list of numbers, while assembling the parser from individual reusable components.

2.1 The Fail Effect

Effect signatures provide the interface of effect operations, but not their implementation. In Effekt, effect signatures are declared as follows:

```
effect Fail[A](msg: String): A
```

The effect operation `Fail` aborts the current computation with a given message. It is polymorphic in its return type `A` so we can use it in any expression position. The following effectful function converts a string to an integer. It uses the `Fail` effect to signal that the conversion failed.

```
def stringToInt(str: String): Int / { Fail } = toInt(str) match {
  case Some(n) ⇒ n
  case None() ⇒ do Fail("cannot convert input to integer")
}
```

In case the optional value returned by the builtin function `toInt` is `None()`, we use the effect operation `Fail` to signal an error. We can freely compose programs that use effects. For instance, we can use the function `stringToInt` to convert and then add two numbers:

```
def perhapsAdd(): Int / { Fail } = stringToInt("1") + stringToInt("2")
```

The type of function `perhapsAdd` communicates that it requires the calling context to handle the `Fail` effect, although `perhapsAdd` does not use the `Fail` effect itself. This requirement arises from the uses of `stringToInt`. Handling effects is similar to handling exceptions:

```
try { perhapsAdd() } with Fail { (msg) ⇒ 0 }
```

In case any conversion fails, the effect handler for the `Fail` effect returns `0` as a default value. The type of the program is `Int / {}`, it has no unhandled effects, and we can run it to get the result `3`.

Capability Passing. Programs in Effekt are translated into a core calculus (System Ξ, Section 4) that explicitly passes handler implementations to their use site. We refer to these explicitly passed handler implementations as *capabilities*. Handlers, like the one for `Fail`, introduce capabilities,

which are then passed as additional arguments to effectful functions. This can be seen in the translation of the above example:

```
handle { Fail ⇒ perhapsAdd( Fail ) } with { (msg, resume) ⇒ 0 }
```

Handling `Fail` introduces an equally named capability, which is passed to `perhapsAdd`. Section 4.3 will go into more detail, for now it suffices to understand that the `Fail` capability contains the handler implementation.

2.2 The Next Effect

Effect handlers generalize exceptions and can express many more effects. Another example is the `Next` effect that we will use to work with a pull-based stream of string values:

```
effect Next(): String
```

Using `Next` and `Fail`, we can express a parser that recognizes a number in the input stream:

```
def number() : Int / { Next, Fail } = stringToInt(do Next())
```

The return type of `number` signals that we can only call it in a context that provides implementations for both `Next` and for `Fail`. We can handle the `Next` effect by always returning `"42"`:

```
def always42[R] { prog: () ⇒ R / { Next } }: R / {} =
  try { prog() } with Next { () ⇒ resume("42") }
```

This handler implementation illustrates the additional power of effect handlers over exception handlers: in an effect handler we can call `resume` to transfer control back to the call-site of the effect operation. While the handler for `Fail` did not use resume, in this example we *resume* the computation with `"42"`. Since `Next` returns a string, `resume` has type `String ⇒ R / {}`. The type signature of $\texttt{always}_{42}$ communicates that it will handle the `Next` effect on `prog`. The empty effect set in the return type signals that $\texttt{always}_{42}$ itself does not require any effects. Since it is polymorphic in the result type `R`, we know that it will call `prog` and handle the `Next` effect. Under our contextual reading, the block passed to $\texttt{always}_{42}$ can have additional effects that need to be handled at the call-site of $\texttt{always}_{42}$. However, the implementation of $\texttt{always}_{42}$ cannot interfere with these additional effects.

```
try { always42 { number() } } with Fail { (msg) ⇒ 0 }
```

In this example, we handle the two effects `Next` and `Fail` that `number` requires with different handlers. Running it results in the integer `42`. Again, inspecting the translated program makes clear how effect types correspond to *requirements* on the caller:

```
handle { Fail ⇒ always42({ Next ⇒ number( Next , Fail ) }) } with { (msg, resume) ⇒ 0 }
```

Now, two separate capabilities for `Next` and `Fail` are passed to the effectful function `number`. Note how the Next capability is introduced by handler $\texttt{always}_{42}$.

Effect handlers grant flexibility in the interpretation of effect operations. We can define another handler for the `Next` effect that reads from a given list.

```
def feed[R](input: List[String]) { prog: () ⇒ R / { Next } } : R / { Fail } = {
  var remaining = input;
  try { prog() } with Next { () ⇒ remaining match {
    case Nil() ⇒ do Fail("End of input")
    case Cons(element, rest) ⇒ remaining = rest; resume(element)
  }}
}
```

This alternative handler shows two interesting things: Firstly, it itself uses the `Fail` effect to signal an unexpected end of the input stream. Secondly, it uses a mutable variable `remaining` to keep track of the position in the input stream. Mutable variables in Effekt are conceptually stack allocated, which guarantees a well-defined interaction with control effects (Section 6).

2.3 The Choice Effect

The handler for `Fail` discarded the resumption, the handler for `Next` called it exactly once. The third and final example effect illustrates that it can be useful to call the resumption *more than once*. For this, we define the effect `Choice`, which returns a boolean to model the outcome of a (potentially) non-deterministic choice:

```
effect Choice(): Boolean
```

In the Effekt language, we can mix effect operations with other imperative language constructs like loops and references. For example, we can define a higher-order function `many` that calls a given program an unknown number of times, controlled by the `Choice` effect:

```
def many { prog: () ⇒ Unit / {} }: Unit / { Choice } = while (do Choice()) { prog() }
```

We use `many` to define a parser that reads arbitrarily many numbers and adds them. Note how using `many` feels as natural as using the built-in control operator `while`:

```
def numbers() = { var res = 0; many { res = res + number() }; res }
```

The inferred return type of `numbers` is `Int / { Choice, Fail, Next }`. Different handlers for `Choice` correspond to different search strategies. One possible example handler performs a backtracking search to find the first success:

```
def backtrack[R] { prog: () ⇒ R / { Fail, Choice } }: Result[R] / {} =
  try { Success(prog()) }
  with Fail { (msg) ⇒ Failure(msg) }
  with Choice { () ⇒ resume(true) match {
    case Failure(msg) ⇒ resume(false)
    case Success(res) ⇒ Success(res)
  }}
```

The handler uses a data type that represents the potentially negative outcome of the search:

```
type Result[R] { Success(res: R); Failure(msg: String) }
```

Handler `backtrack` handles *two* effects: `Fail` and `Choice`. At each choice, it first resumes with `true` and in case of failure resumes a second time with `false`. Any use of the `Fail` effect aborts the current search path with `Failure`.

2.4 Parsing

Parsers like `numbers` use the effects `Fail`, `Next`, and `Choice` that we group under an effect alias:

```
effect Parser = { Fail, Next, Choice }
```

To handle the `Parser` effect, we simply reuse the handler implementations from this section:

```
def parse[R](input: List[String]) { prog: () ⇒ R / Parser }: Result[R] / {} =
  backtrack { feed(input) { prog() } }
```

A parser handles `Next` by reading from the given list of strings. It handles all failures in `prog` and in `feed` using the implementation of backtracking search. By nesting `feed` inside of `backtrack`,

the position in the input stream is automatically correctly backtracked when a choice is resumed a second time (Section 6). Running the program `parse(["1", "2"]) { numbers() }` outputs the number 3.

Section conclusion. Effect handlers generalize exception handlers and offer additional expressivity. This way, the advanced control flow of programs like `numbers` can be modularly described in user defined combinators. Our backtracking implementation corresponds roughly to a hand-written recursive descent parser with the advantage that the decision for a parsing algorithm is not hard coded into parsers like `numbers`. The imperative parser combinators can easily be combined with other effects like mutable state, exceptions, or even let-insertion [Yallop 2017]. Abstractions like the `many` combinator can be shared in a library and reused across different domains.

3 THE LANGUAGE Effekt

In this section, we formally present the syntax and type system of the Effekt language. Section 4 then introduces the core calculus System Ξ, and Section 5 defines the semantics of Effekt by translation into System Ξ.

3.1 Syntax

Figure 1 defines the syntax of Effekt. Like other languages with effect handlers [Hillerström et al. 2017] it is presented in fine-grain call-by-value [Levy et al. 2003]. That is, we syntactically distinguish expressions, which cannot have control effects from statements that can have control effects.

Expressions. Effect safety of Effekt rests on the property that all functions (blocks) are second class. Consequently, blocks are syntactically *neither* values *nor* expressions. Only primitive constants are values. Similarly, we distinguish syntactically between variables that stand for values (x, y, ...) and variables that stand for blocks (f, g, ...). As usual, we follow Barendregt [1992] and require that all expression variables, block variables, and operation names are globally unique.

Statements. We sequence two statements with **val** $x = s_0; s_1$, where the result of s_0 will be bound to the variable x in s_1. The syntactic form **def** $f(\overline{x : \tau}, \overline{g : \sigma}) : \tau / \varepsilon = s_0; s$ defines a *block* f, binding a fixed, but arbitrary number of value variables x as well as block variables g. As seen in Section 1, effect types can influence the operational semantics and we thus require the types of arguments and the return type (*e.g.*, τ / ε) to be annotated. Calling blocks is denoted $f(\overline{e}, \overline{g})$, providing potentially multiple value arguments as expressions e and block arguments as block variables g. Without loss of generality, we do not allow passing *anonymous* blocks and require that all blocks are named, before passing them to a call. This convention significantly simplifies the presentation of the typing rules and the translation. Effect declarations are statements of the form **effect** $F(x : \tau) : \tau; s$, which means that effects can be declared locally. Effect calls (*i.e.*, **do** $F(e)$) only take a single expression argument. While the restriction to one argument is insignificant, it is important that effect operations only take expressions as arguments and never blocks. Otherwise blocks could escape their scope through the effect operation [Brachthäuser et al. 2020a], violating effect safety. Finally, the statement **try** { s } **with** F { $(x : \tau) \Rightarrow s'$ } expresses that effect calls to F in the handled statement s will be handled by the handler F { $(x : \tau) \Rightarrow s'$ }. The special block variable `resume` is available in the handler implementation s'.

Types. The meta variable τ describes *value types*, which we use to type expressions. The meta variable σ describes *block types*, which we use to type blocks. Expressions are first-class, while blocks are second-class [Osvald et al. 2016]. A block type (*i.e.*, $(\overline{\tau}, \overline{\sigma}) \to \tau / \varepsilon$) takes expressions of types $\overline{\tau}$ and blocks of types $\overline{\sigma}$ as parameters. The return type τ of a block indicates that only values (and not blocks) can be returned. The block type also mentions the effects ε that need to be

Syntax:

Statements	s	::=	**val** $x = s;\ s$	sequencing
		|	e	expressions
		|	**def** $f(\overline{x : \tau}, \overline{g : \sigma}) : \tau \,/\, \varepsilon = s;\ s$	block definition
		|	$f(\overline{e}, \overline{g})$	block call
		|	**effect** $F(x : \tau) : \tau;\ s$	effect declaration
		|	**do** $F(e)$	effect call
		|	**try** $\{\ s\ \}$ **with** $F\ \{\ (x : \tau) \Rightarrow s\ \}$	effect handling
Expressions	e	::=	$x \mid v$	
Expression Values	v	::=	() | 0 | 1 | ... | true | false | ...	primitives

Syntax of Types:

Value Types	τ	::=	Int | Bool | ...	Value Environment	Γ	::=	$\emptyset \mid \Gamma,\ x : \tau$
Block Types	σ	::=	$(\overline{\tau}, \overline{\sigma}) \to \tau \,/\, \varepsilon$	Block Environment	Δ	::=	$\emptyset \mid \Delta,\ f : \sigma$
Effect Sets	ε	::=	$\{\ F_1, ..., F_n\ \}$	Effect Environment	Σ	::=	$\emptyset \mid \Sigma,\ F : \tau \to \tau$

Names:

Expression Variables $x, y \in$ x, y Block Variables $f, g \in$ f, g Operations $F \in$ Fail, Choice, ...

Fig. 1. Syntax of the source language Effekt.

handled by the caller. Effects $\varepsilon = \{ F_1, ..., F_n \}$ are (closed) sets of operation names F_i. This is different from languages that base their effect systems on row polymorphic records where effect operations can occur multiple times [Leijen 2017b] or effects are annotated with presence/absence information [Hillerström and Lindley 2016]. Modeling effects as sets greatly simplifies typing as no special unification rules are needed [Leijen 2005].

3.2 Typing

Figure 2 defines the typing rules of Effekt. To understand the type system, it is important to recall the interpretation of effect types in Effekt:

Effect types express which capabilities a computation requires from its context.

This intuition will be useful when we discuss the details of the typing rules. There are two judgments, one for expressions and one for statements.

3.2.1 Expression Typing. The judgment for expressions $\Gamma \vdash e : \tau$ assigns a value type τ to an expression e in value environment Γ. Typing of expressions only requires a value environment, since expressions cannot mention any blocks or effects. Furthermore, since expressions do not have control effects, expression typing computes a value type τ without any effects. The typing rules for expressions are completely standard.

3.2.2 Statement Typing. The judgment for statements $\Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon$ computes a value type τ and a set of required capabilities ε for the statement s. It uses three environments, a value environment Γ, a block environment Δ, and an effect environment Σ. Rule Val types sequencing of statements. It accumulates all required capabilities of the binding s_0 and the body s_1 by taking the union of the corresponding effect sets ε_0 and ε_1. Rule Expr types an expression statement by assigning the empty set of effects. Rule Def types block definitions. It is a bit more involved and

Statement Typing. $\boxed{\Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon}$ (↑ ↑ ↑ ↑ ↓ ↓)

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash s_0 : \tau_0 \mid \varepsilon_0 \quad \Gamma, x : \tau_0 \mid \Delta \mid \Sigma \vdash s_1 : \tau_1 \mid \varepsilon_1}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{val}\ x = s_0; s_1 : \tau_1 \mid \varepsilon_0 \cup \varepsilon_1}\ [\textsc{Val}] \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \mid \Delta \mid \Sigma \vdash e : \tau \mid \emptyset}\ [\textsc{Expr}]$$

$$\frac{\Gamma, \overline{x : \tau} \mid \Delta, \overline{g : \sigma} \mid \Sigma \vdash s_0 : \tau_0 \mid \varepsilon_0' \qquad \Gamma \mid \Delta, f : (\overline{\tau}, \overline{\sigma}) \to \tau_0 \,/\, \varepsilon_0 \mid \Sigma \vdash s : \tau \mid \varepsilon}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{def}\ f(\overline{x : \tau}, \overline{g : \sigma}) : \tau_0 \,/\, \varepsilon_0 = s_0; s : \tau \mid (\varepsilon_0' \setminus \varepsilon_0) \cup \varepsilon}\ [\textsc{Def}]$$

$$\frac{\overline{\Gamma \vdash e : \tau} \qquad \overline{\Delta(g) = \sigma} \qquad \Delta(f) = (\overline{\tau}, \overline{\sigma}) \to \tau \,/\, \varepsilon}{\Gamma \mid \Delta \mid \Sigma \vdash f(\overline{e}, \overline{g}) : \tau \mid \varepsilon}\ [\textsc{BlockCall}]$$

$$\frac{\Gamma \mid \Delta \mid \Sigma, F : \tau_1 \to \tau_0 \vdash s_2 : \tau_2 \mid \varepsilon_2 \qquad F \notin \mathsf{ftv}(\varepsilon_2)}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{effect}\ F(x_1 : \tau_1) : \tau_0; s_2 : \tau_2 \mid \varepsilon_2}\ [\textsc{Effect}]$$

$$\frac{\Sigma(F) = \tau_1 \to \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{do}\ F(e_1) : \tau_0 \mid \{ F \}}\ [\textsc{EffectCall}]$$

$$\frac{\begin{array}{c}\Sigma(F) = \tau_1 \to \tau_0 \qquad \Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon \\ \Gamma, x_1 : \tau_1 \mid \Delta, \mathsf{resume} : (\tau_0) \to \tau \,/\, \emptyset \mid \Sigma \vdash s' : \tau \mid \varepsilon_0\end{array}}{\Gamma \mid \Delta \mid \Sigma \vdash \textbf{try}\ \{ s \}\ \textbf{with}\ F\ \{ (x_1 : \tau_1) \Rightarrow s' \} : \tau \mid (\varepsilon \setminus \{ F \}) \cup \varepsilon_0}\ [\textsc{Try}]$$

Expression Typing. $\boxed{\Gamma \vdash e : \tau}$ (↑ ↑ ↓)

$$\frac{}{\Gamma \vdash n : \mathsf{Int}}\ [\textsc{Lit}] \qquad \frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ [\textsc{Var}]$$

Fig. 2. Typing rules of the source language Effekt.

requires some explanation. The capabilities a block requires can be provided in two different ways. Firstly, the block can mention a required effect in its type, which means that the effect is handled *dynamically* and the capability needs to be provided by the caller. Secondly, all capabilities that are not part of the annotated type need to be provided by the context at the *definition site* of the block. This can be seen in analogy to term-level variables: free variables in a function body can either be bound as parameters of the function, or they are bound in the context of the definition site. Hence, we call this form of handling *lexical handling*. Rule Def embodies the essence of contextual effect polymorphism: using type annotations, programmers can control which effects are handled at the call-site (*i.e.*, ε_0), and which effects are *free* (*i.e.*, $\varepsilon_0' \setminus \varepsilon_0$) and need to be handled at the definition site of a block. Operationally, as we will see in Section 5, the block will close over the free effects. Since all complications are part of rule Def, typing block calls (rule BlockCall) takes a familiar form, simply checking whether the argument types conform to the annotated parameter types. The decision whether effects of (anonymous) block arguments are handled dynamically or lexically is clear from the types at their definitions.

Example 3.1. The following example illustrates how the Def rule enables contextual effect polymorphism:

```
def optionally { prog: () → Int / { Fail } }: Option[Int] / {} = ...;
optionally { () ⇒ if (do Choice()) do Fail("failed") else 42 }
```

We desugar this example to bind the block to a fresh name anon with exactly the type of the block parameter of optionally.

```
def optionally { prog: () → Int / { Fail } }: Option[Int] / {} = ...;
def anon(): Int / { Fail } = if (do Choice()) do Fail("failed") else 42;
optionally(anon)
```

The example program has the overall type Option[Int] / { Choice }. The required capability Fail is provided by function optionally, while Choice is free and has to be provided by the context of the definition of anon.

The last three rules are concerned with effect declaration, use, and handling. Rule Effect extends the effect environment Σ, bringing the effect operation F into scope. The side-condition $F \notin \mathsf{ftv}(\varepsilon_2)$ corresponds to the standard check that type variables should not leave the scope in which they are defined [Eisenberg et al. 2018]. Symmetrically, rule EffectCall requires the effect to be lexically in scope when an effect operation is used. Interestingly, in our capability-oriented formalization, this simple check suffices to express locally defined effects, where other languages require sophisticated use of existential quantification [Biernacki et al. 2019]. Lastly, rule Try types handling of effects. The handled statement s is assigned return type τ and effects ε. In Effekt, after typing s, the handled effect is subtracted from the resulting set of effects ε. This, again, is an important difference compared to languages based on row polymorphism [Leijen 2017b] where, by unification, the effect type of s would necessarily include F. The body of the handler s' can assume that the variable x_1 has type τ_1 and that the block variable resume has type $(\tau_0) \longrightarrow \tau \,/\, \emptyset$. The latter might come with surprise: one might expect that the continuation still has effects. However, recalling our contextual reading of effects, we can see that resume is merely contextually pure. All effects in resume are handled outside the corresponding **try** statement. In particular, the body of the operation clause does not have to (and even cannot) handle any effects in resume. Typing the continuation with the empty set of effects is safe since it is a block and cannot leave the scope of its definition. Finally, the resulting set of effects is the set of effects ε of the handled statement, without F, but including all effects used by the effect operation ε_0.

4 THE CORE LANGUAGE System Ξ

To specify the semantics of Effekt, we translate it to a core language: System Ξ. This section presents its syntax and type system, sketches its operational semantics, and states semantic soundness. Section 5 then defines the translation from Effekt to System Ξ and shows it preserves well-typedness. Effect safety of Effekt follows as a corollary. The full description of the operational semantics, type system, as well as detailed proofs can be found in the extended technical report [Brachthäuser et al. 2020b]. We have mechanized the type system and operational semantics of System Ξ in the dependently typed programming language Idris [Brady 2013]. Our implementation of Effekt closely follows the translation presented in Section 5.

4.1 Syntax

Figure 3a defines the syntax of System Ξ. Like Effekt, the core language is in fine-grain call-by-value. Also like Effekt, it distinguishes expressions e and blocks b. Unlike Effekt, however, the core language supports effect handlers in *explicit capability-passing style*. That is, effect operations are represented by blocks, which are introduced by the corresponding handler and passed as additional arguments. As a consequence, System Ξ does not distinguish between named blocks (that is, function definitions), anonymous blocks, and effect operations – all three are represented by the syntactic category of blocks b. Blocks b can either be block variables f or block values w of

Syntax of Terms:

Statements	s	::=	$\textbf{val}\ x\ =\ s;\ s$	sequencing
		|	e	expressions
		|	$\textbf{def}\ f\ =\ b;\ s$	block definition
		|	$b(\overline{e},\ \overline{b})$	block call
		|	$\textbf{handle}\ \{\ F \Rightarrow s\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s\ \}$	handler
Expressions	e	::=	$x \mid v$	
Expression Values	v	::=	$() \mid 0 \mid 1 \mid ... \mid \mathsf{true} \mid \mathsf{false} \mid ...$	constants
Blocks	b	::=	$f \mid w$	
Block Values	w	::=	$\{\ (\overline{x :\ \tau},\ \overline{f :\ \sigma}) \Rightarrow s\ \}$	

Syntax of Types:

Value Types	τ	::=	$\mathsf{Int} \mid \mathsf{Bool} \mid ...$	Value Environment	Γ	::=	$\emptyset \mid \Gamma,\ x :\ \tau$
Block Types	σ	::=	$(\overline{\tau},\ \overline{\sigma}) \to \tau$	Block Environment	Δ	::=	$\emptyset \mid \Delta,\ f :\ \sigma$

Names:

Expression Variables $x,\ y\ \in \mathsf{x},\ \mathsf{y}$ Block Variables $f,\ g,\ k,\ F\ \in \mathsf{f},\ \mathsf{g},\ \mathsf{k},\ \mathsf{Fail},\ \mathsf{Choice},\ ...$

(a) Syntax of System Ξ.

Type Rules:

Statement Typing. $\boxed{\Gamma \mid \Delta \vdash\ s\ :\ \tau}$

$$\frac{\Gamma \mid \Delta \vdash\ s_0\ :\ \tau_0 \quad \Gamma,\ x :\ \tau_0 \mid \Delta \vdash\ s_1\ :\ \tau_1}{\Gamma \mid \Delta \vdash\ \textbf{val}\ x\ =\ s_0;\ s_1\ :\ \tau_1}\ [\textsc{Val}]$$

$$\frac{\Gamma \vdash\ e\ :\ \tau}{\Gamma \mid \Delta \vdash\ e\ :\ \tau}\ [\textsc{Expr}] \qquad \frac{\Gamma \mid \Delta \vdash\ b\ :\ \sigma \quad \Gamma \mid \Delta,\ f\ :\ \sigma \vdash\ s\ :\ \tau}{\Gamma \mid \Delta \vdash\ \textbf{def}\ f\ =\ b;\ s\ :\ \tau}\ [\textsc{Def}]$$

$$\frac{\Gamma \mid \Delta \vdash\ b\ :\ (\overline{\tau},\ \overline{\sigma}) \to \tau_0 \quad \overline{\Gamma \vdash\ e\ :\ \tau} \quad \overline{\Gamma \mid \Delta \vdash\ b\ :\ \sigma}}{\Gamma \mid \Delta \vdash\ b(\overline{e},\ \overline{b})\ :\ \tau_0}\ [\textsc{Call}]$$

$$\frac{\Gamma \mid \Delta,\ F :\ \tau_1 \to \tau_0 \vdash\ s\ :\ \tau \qquad \Gamma,\ x :\ \tau_1 \mid \Delta,\ k :\ \tau_0 \to \tau \vdash\ s'\ :\ \tau}{\Gamma \mid \Delta \vdash\ \textbf{handle}\ \{\ F \Rightarrow s\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s'\ \}\ :\ \tau}\ [\textsc{Handle}]$$

Block Typing. $\boxed{\Gamma \mid \Delta \vdash\ b\ :\ \sigma}$

$$\frac{\Delta(f)\ =\ \sigma}{\Gamma \mid \Delta \vdash\ f\ :\ \sigma}\ [\textsc{BlockVar}] \qquad \frac{\Gamma,\ \overline{x :\ \tau} \mid \Delta,\ \overline{f :\ \sigma} \vdash\ s_0\ :\ \tau_0}{\Gamma \mid \Delta \vdash\ \{\ (\overline{x :\ \tau},\ \overline{f :\ \sigma}) \Rightarrow s_0\ \}\ :\ (\overline{\tau},\ \overline{\sigma}) \to \tau_0}\ [\textsc{Block}]$$

Expression Typing. $\boxed{\Gamma \vdash\ e\ :\ \tau}$

$$\frac{\Gamma(x)\ =\ \tau}{\Gamma \vdash\ x\ :\ \tau}\ [\textsc{Var}] \qquad \frac{}{\Gamma \vdash\ n\ :\ \mathsf{Int}}\ [\textsc{Lit}]$$

(b) Type system of System Ξ.

Fig. 3. Syntax and type system of System Ξ.

the form $\{\ (\overline{x : \tau}, \overline{f : \sigma}) \Rightarrow s\ \}$. Note how block variables (*e.g.*, Choice or Fail) in System Ξ may have the names of effect operations in Effekt. Local blocks are defined with **def** $f = b;\ s$, binding block b to the name f in scope of the statement s. Block calls in System Ξ, of the form $b(\overline{e}, \overline{b})$, subsume block and effect calls of Effekt. Arguments can be an arbitrary number of value arguments and block arguments. Finally, the handle statement **handle** $\{\ F \Rightarrow s\ \}$ **with** $\{\ (x,\ k) \Rightarrow s'\ \}$ binds the block variable F in the handled statement s. The value parameter x and the continuation block k are bound in the handler implementation s'.

4.2 Typing

The typing rules of System Ξ are defined in Figure 3b. Like the source language, System Ξ distinguishes between two kinds of types: value types τ and block types σ. Importantly, block types now do not mention any effects, but only map value- and block parameter types to a resulting value type τ. Furthermore, inspecting the type system of System Ξ, we can see that it does not include an effect system. Effect safety is simply established by treating blocks as second class. The type system of System Ξ has three judgments, one for each syntactic category. Statements and blocks are typed against two environments: environment Γ for value variables and environment Δ for block variables. Since effects are translated to blocks, the signature environment Σ is not required anymore. Besides distinguishing between values and blocks and using separate environments, the typing rules for sequencing (Val), expressions in statement position (Expr), block definitions (Def), and block calls (Call) are completely standard. They correspond to the rules of Effekt but without any tracking of effects. Rule Handle is central to the calculus. We handle statement s with the handler implementation s'. The handler introduces a capability and binds it to the operation name F, which is brought into scope as a block variable in the handled statement s. In the handler implementation s', the parameter of the effect operation x has type τ_1 and the continuation k is an ordinary block variable of type $\tau_0 \to \tau$. The (answer) type τ appears four times in this rule: As the return type of the overall statement, the return type of the handled statement, the result type of the continuation, and the return type of the handler implementation – all have to agree.

4.3 Operational Semantics

We give the semantics of System Ξ as a small-step operational semantics using evaluation contexts [Wright and Felleisen 1994]. To allow capturing and resuming continuations, the semantics of System Ξ follows the generative semantics presented by Biernacki et al. [2020], who in turn present a variant of multi-prompt delimited control [Gunter et al. 1995]. Like previous presentations of effect handlers [Kammar et al. 2013], capturing a continuation removes the corresponding delimiter and resuming reinstalls the delimiter. This corresponds to a multi-prompt variant of the control operator shift_0 [Danvy and Filinski 1989]. Our presentation requires two additional runtime constructs that only appear during evaluation: delimiters and capabilities.

Labels. Both runtime constructs refer to unique runtime labels l, generated during reduction. We only require that labels can be compared for equality and that we are able to generate fresh labels at runtime. In this paper, we represent concrete labels as hexadecimal hashes (*e.g.*, `@a5f`) to highlight that they are created at runtime.

Delimiters. The additional statement $\#_l\ \{\ s\ \}$ represents a *delimiter* that delimits a statement s at a given label l (or *prompt* in the terminology of Felleisen [1988], Sitaram [1993], and Gunter et al. [1995]).

Capabilities. The additional block value $\mathbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \}$ represents a *capability*, which is a pair of a label l and a handler implementation s [Brachthäuser and Schuster 2017]. Calling a

capability captures the stack segment up to the next dynamically enclosing delimiter for the label l, reifies it as a continuation, and binds it to k.

4.3.1 Reduction Rules. The presentation of the operational semantics follows Gunter et al. [1995] and is based on *delimited evaluation contexts* H_l where the label l does not appear in any delimiters in H_l. It is used to guarantee that captured continuations are always delimited by the dynamically closest delimiter for a label.

Handling introduces delimiters. Rule *(handle)* creates a fresh runtime label l, delimits the handled statement s with this label, and substitutes a capability that refers to l for the block variable F.

$$\textit{(handle)}\quad \textbf{handle}\ \{\ F \Rightarrow s\ \}\ \textbf{with}\ \{\ (x,\ k) \Rightarrow s'\ \} \longrightarrow \#_l\ \{\ s[F \mapsto \textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s'\ \}]\ \} \qquad l\ \textit{fresh}$$

Since the label is fresh, the capability is only valid in the dynamic region delimited by $\#_l$. Calling the capability outside of the region will lead to a stuck term. Our semantics is *generative*: reducing the same **handle** statement twice will introduce two distinct runtime labels [Biernacki et al. 2020].

Capabilities capture the continuation. The most interesting rule *(cap)* captures part of the context:

$$\textit{(cap)}\quad \#_l\ \{\ \mathsf{H}_l[\ (\textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})(v)\ ]\ \} \longrightarrow s[x \mapsto v,\ k \mapsto \{\ y \Rightarrow \#_l\ \{\ \mathsf{H}_l[\ y\ ]\ \}\ \}]$$

The application of a capability to a value (*e.g.*, $(\textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s\ \})(v)$) itself is *not* a redex. This highlights the essence of control effects: they depend on (and modify) the context they are evaluated in. The application of a capability with label l is only meaningful in a context, which is delimited at label l. This becomes visible in rule *(cap)*, where the delimiter $\#_l$, the delimited context H_l, and the capability application together form a redex. We reify this context as a continuation and substitute it (as well as the argument v) in the body of the handler implementation. Effect safety means that applications of a capability with label l only occur in a context with a delimiter at l (Theorem 4.3).

Only values can leave delimiters. Once a statement is reduced to a value, delimiters are discarded:

$$\textit{(ret)}\quad \#_l\ \{\ v\ \} \longrightarrow v$$

Since blocks (and capabilities) are no expression values, they cannot be returned.

Example 4.1. The following example illustrates the operational semantics of capturing continuations. We assume an effect operation $\Delta(\mathsf{Yield}) = \mathsf{Int} \to \mathsf{Int}$.

$$\textbf{handle}\ \{\ \mathsf{Yield} \Rightarrow \textbf{val}\ \mathsf{x} = \mathsf{Yield}(20);\ \mathsf{x} * 2\ \}\ \textbf{with}\ \{\ (\mathsf{x},\ \mathsf{k}) \Rightarrow \mathsf{k}(\mathsf{x} + 1)\ \}$$

Reducing **handle** introduces a fresh label (*e.g.*, `@a1`) and uses it to delimit the handled program. It also introduces a capability and substitutes it for Yield:

$$\#_{\texttt{@a1}}\ \{\ \textbf{val}\ \mathsf{x} = (\textbf{cap}_{\texttt{@a1}}\ \{\ (\mathsf{x},\ \mathsf{k}) \Rightarrow \mathsf{k}(\mathsf{x} + 1)\ \})(20);\ \mathsf{x} * 2\ \}$$

Applying rule *(cap)*, we obtain (captured stack segment highlighted in *gray*):

$$\mathsf{k}(20 + 1) \quad \text{where } \mathsf{k} = \{\ \mathsf{y} \Rightarrow \#_{\texttt{@a1}}\ \{\ \textbf{val}\ \mathsf{x} = \mathsf{y};\ \mathsf{x} * 2\ \}\ \}$$

Further reducing the application results in

$$\#_{\texttt{@a1}}\ \{\ \textbf{val}\ \mathsf{x} = 21;\ \mathsf{x} * 2\ \}$$

where we proceed to reduce under the delimiter to obtain $\#_{\texttt{@a1}}\ \{\ 42\ \}$, and finally remove the delimiter to get the result 42. As can be seen from the example, capturing the continuation removes the corresponding delimiter $\#_{\texttt{@a1}}$ and calling the continuation reinstalls it. This treatment of delimiters together with capability passing models *deep handlers* [Kammar et al. 2013].

Translation of Block Types and Effect Types:

$$
\begin{array}{lcl}
\mathcal{T}[\![\ (\overline{\tau},\ \overline{\sigma}) \to \tau_0\ /\ \{\ F_1,\ ....\ F_n\ \}\ ]\!] & = & (\overline{\tau},\ \overline{\mathcal{T}[\![\sigma]\!]},\ \mathcal{T}[\![F_1]\!],\ ...,\ \mathcal{T}[\![F_n]\!]) \to \tau_0 \\
\mathcal{T}[\![\ F\ ]\!] & = & (\tau_1) \to \tau_0 \\
\quad \text{where} \quad \Sigma(F) = \tau_1 \to \tau_0 & & \\
\mathcal{T}[\![\ \{F_1,\ ...,\ F_n\}\ ]\!] & = & F_1\ :\ \mathcal{T}[\![F_1]\!],\ ...,\ F_n\ :\ \mathcal{T}[\![F_n]\!]
\end{array}
$$

Translation of Statements:

$$
\begin{array}{lcl}
\mathcal{S}[\![\ \textbf{val}\ x\ =\ s_0;\ s_1\ ]\!] & = & \textbf{val}\ x\ =\ \mathcal{S}[\![\ s_0\ ]\!];\ \mathcal{S}[\![\ s_1\ ]\!] \\
\mathcal{S}[\![\ e\ ]\!] & = & e \\
\mathcal{S}[\![\ \textbf{def}\ f(\overline{x},\ \overline{g})\ :\ \tau_0\ /\ \varepsilon_0\ =\ s_0;\ s\ ]\!] & = & \textbf{def}\ f\ =\ \{\ (\overline{x},\ \overline{g},\ F_1,\ ...,\ F_n) \Rightarrow \mathcal{S}[\![\ s_0\ ]\!]\ \};\ \mathcal{S}[\![\ s\ ]\!] \\
\quad \text{where} \quad \varepsilon_0\ =\ \{\ F_1,\ ....\ F_n\ \} & & \\
\mathcal{S}[\![\ f(\overline{e},\ \overline{g})\ ]\!] & = & f(\overline{e},\ \overline{g},\ F_1,\ ...\ F_n) \\
\quad \text{where} \quad f\ :\ (\overline{\tau},\ \overline{\sigma}) \to \tau_0\ /\ \{\ F_1,\ ...\ F_n\ \} & & \\
\mathcal{S}[\![\ \textbf{effect}\ F(x_1\ :\ \tau_1)\ :\ \tau_0;\ s\ ]\!] & = & \mathcal{S}[\![\ s\ ]\!] \\
\mathcal{S}[\![\ \textbf{do}\ F(e_1)\ ]\!] & = & F(e_1) \\
\mathcal{S}[\![\ \textbf{try}\ \{\ s\ \}\ \textbf{with}\ \{\ F(x) \Rightarrow s'\ \}\ ]\!] & = & \textbf{handle}\ \{\ F \Rightarrow \mathcal{S}[\![\ s\ ]\!]\ \}\ \textbf{with}\ \{\ (x,\ \text{resume}) \Rightarrow \mathcal{S}[\![\ s'\ ]\!]\}
\end{array}
$$

Fig. 4. Translation of Effekt to System Ξ – we assume a canonical ordering of effects in ε.

4.4 Soundness

In our mechanized formalization, we represent System Ξ terms by their typing derivations [Benton et al. 2012] and show progress constructively by implementing the semantics as a total step function. One important class of stuck terms are capability applications without a corresponding delimiter.

Definition 4.2 (Undelimited Label). A statement s contains an undelimited label l, if it has the form $\mathsf{H}_l[(\textbf{cap}_l\ \{\ (x,\ k) \Rightarrow s'\ \})(v)]$.

In our operational semantics, reducing a redex never produces a newly undelimited label. In the type system for System Ξ extended with runtime constructs, we add an additional *label context* Ξ to the typing judgement, which now has the form $\Gamma \mid \Delta \mid \Xi \vdash s : \tau$. All typing rules in Figure 3b ignore Ξ and simply pass it to the premises. We use this label context in our proof to formally capture this invariant.

Starting from an empty label context, closed and well-typed System Ξ programs either are values or we can take a step. Here the relation $\longmapsto$ describes congruence, that is, reduction under a context.

Theorem 4.3 (Progress of System Ξ). *If* $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$, *then* s *is a value* v *or* $s \longmapsto s'$.

Our mechanized formalization establishes preservation by indexing System Ξ programs with their type. Performing a reduction step on a statement preserves its type:

Theorem 4.4 (Preservation of System Ξ). *If* $\emptyset \mid \emptyset \mid \emptyset \vdash s : \tau$ *and* $s \longmapsto s'$ *then* $\emptyset \mid \emptyset \mid \emptyset \vdash s' : \tau$.

In particular, reduction also preserves the (empty) label context. That is, from progress and preservation follows effect safety: programs are never stuck on an undelimited label.

5 TRANSLATION OF Effekt TO System Ξ

Having introduced both Effekt and System Ξ formally, we now show how to make the flow of capabilities explicit by translating Effekt into System Ξ, *i.e.* into explicit capability-passing style. The translation is type directed and operates on typing derivations. Intuitively, where in Effekt the effect types indicate that a computation requires capabilities to be available in its context,

in System Ξ we explicitly pass such capabilities as additional arguments. Figure 4 defines the translation. The translation is defined on types and on statements. We neither translate expressions, nor their types, since the language of expressions is the same in Effekt and System Ξ.

Translation of types. We translate blocks that require a set of effects $\{ F_1, \ldots, F_n \}$ to blocks that receive n additional block arguments – one for each member of the set. The translation of effect sets to additional arguments can be seen in the translation of block types, of block definitions, and of block calls. Names of operations in Effekt are now names of block variables in System Ξ. For the translation, we assume a canonical ordering of effects in each ε. In our implementation, it suffices to choose an arbitrary but fixed ordering for each type signature. The translation of types extends to environments and sets of effects. In particular, we translate the sets of effects ε of Effekt into block environments Δ of System Ξ translating each effect operation to a binding in the block environment. We assume that names of blocks and names of effect operations are disjoint and no name conflicts arise.

Translation of terms. The translation of block definitions uses type information to add additional capability parameters to the block f. Symmetrically, the translation of application adds additional arguments. Assuming $\Sigma(\mathsf{Fail}) = (\mathsf{String}) \longrightarrow \mathsf{Int}$, the program of Example 3.1 translates to:

```
def optionally = { (prog : ((String) → Int) → Int) ⇒ ... }
def anon = { (Fail : (String) → Int) ⇒ if (Choice()) then Fail("failed") else 42 }
optionally(anon)
```

Effect calls translate to ordinary block calls, where the name of the called block is the same as the name of the effect operation (*e.g.*, Choice or Fail). The effect system of Effekt and the translation guarantee that a block with the name of the effect operation is in scope. Effect types and their declarations disappear during translation. Translating the **try** statement of Effekt into the **handle** statement of System Ξ makes two things explicit: Firstly, the **handle** statement now explicitly binds the capability F as a block in the handled statement s. Secondly, the continuation resume is bound explicitly as a *block variable* in the body s'.

5.1 Well-Typedness Preservation

The translation from Effekt to System Ξ in explicit capability-passing style preserves well-typedness:

Theorem 5.1 (Translation preserves well-typedness).
If $\Gamma \mid \Delta \mid \Sigma \vdash s : \tau \mid \varepsilon$ *, then* $\Gamma \mid \mathcal{T}[\![\Delta]\!] + \mathcal{T}[\![\varepsilon]\!] \mid \emptyset \vdash \mathcal{S}[\![s]\!] : \tau$.

Proof. Straightforward induction over the typing derivations. □

The translated program $\mathcal{S}[\![s]\!]$ is valid under the empty label context $\Xi = \emptyset$, *i.e.* does not contain any undelimited labels. This is obvious as the translation (Figure 4) never introduces any labels, delimiters, or capabilities. Those are only introduced at runtime by reducing **handle** statements.

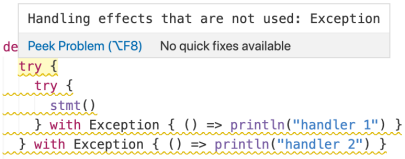
5.2 Semantic Soundness of Effekt

We define the semantics of Effekt as the composition of the translation to System Ξ and the semantics of System Ξ. This presentation emphasizes our capability-based understanding of effects by translating them to explicitly passed blocks. Semantic soundness (that is, effect safety) of Effekt directly follows from preservation of well-typedness (Theorem 5.1) and soundness of System Ξ. We can identify two potential sources of runtime errors that would violate effect safety:

Unhandled effects. Effects in Effekt might be unhandled, that is there is no enclosing effect handler that would handle the effect. In our translation, effect operations are translated to block

```
const usercode =
  handle([{
    "op$Exception": resume ⇒ println("empty line")
  }])(Exception ⇒
    eachLine(someFile, (line) ⇒
      (line == "") ? Exception.op$Exception()
                   : pure(unit)))
```

(a) JavaScript code generated for the running example from Section 1.



(b) Warnings facilitate the development of programs using effects and handlers.

Fig. 5. Practical aspects: compilation output and IDE integration.

calls. Unhandled effects thus correspond to *unbound block variables*. The type-system of System Ξ and well-typedness preservation guarantees that such programs cannot be expressed.

Escaping capabilities. Effects are translated to capabilities and those contain labels. Those capabilities could (accidentally) *leave the scope* of the corresponding delimiter, leading to a runtime error. However, this is ruled out by preservation (Theorem 4.4) of System Ξ. Furthermore, the translation does not introduce any delimiters or uses runtime labels in any other way.

6 PRACTICAL EVALUATION

To evaluate the design of Effekt, we implemented a compiler from Effekt to JavaScript[3]. The evaluation aims to establish feasibility of a language implementation, practically investigate expressivity and severity of the restrictions imposed by the language design, explore the interaction with (seemingly orthogonal) extensions to the language, evaluate the integration with a target language like JavaScript (via foreign function interfaces), practically evaluate effect inference in combination with bidirectional typechecking, and finally further explore the design space and extensions with additional interesting features. Our implementation scales the presented calculus Effekt to a fully fledged language. Among others, it features algebraic data types, (nested) pattern matching, value polymorphism, type directed overloading, local (backtrackable) mutable state, and a foreign function interface to JavaScript. We used the language implementation to develop multiple, medium sized case studies. In this section, we describe the language implementation, discuss differences to the formal presentation of Effekt, and elaborate on our experience in using Effekt to implement the case studies. The language implementation, an implementation of a Visual Studio Code plugin, as well as the documented case studies can be found in the repository accompanying this paper[4].

6.1 Description of the Implementation

Our implementation of Effekt is based on the translation to System Ξ as presented in this paper. Additionally, the compiler then translates System Ξ to a subset of JavaScript using a library implementation of monadic delimited control [Dybvig et al. 2007]. The implementation of the Effekt language spans around 7.000 lines of code, while the monadic JavaScript runtime system only comprises around 150 lines of code.

6.1.1 Generated JavaScript Code. To get an impression of the generated JavaScript code, Figure 5a shows the result of translating the user program of our running example. The JavaScript code is in explicit capability-passing style. The library function `handle` takes the handler implementation and introduces a capability `Exception` in the body, which is passed as second argument. The resulting code also shows the use of the library for monadic delimited control [Brachthäuser et al.

[3] https://effekt-lang.org
[4] https://github.com/effekt-lang/effekt

2020a; Dybvig et al. 2007]. Pure values without control effects are embedded into the monad by calling `pure`. Programs like `usercode` can be executed with `usercode.run()`. Since JavaScript is call-by-value, our library implementation additionally includes a function `delay` that expects a computation without control effects, but with potential side effects of the target language. It defers the side effects until they are forced by corresponding call to `run()`.

6.1.2 IDE Support for Effects and Handlers. For our practical evaluation, we implemented a Visual Studio Code extension. In addition to the usual features of IDEs (like error reporting or jumping to definitions), two features proved to be particularly useful to develop effectful programs: Firstly, while inferring return types (and effects) of functions facilitates refactorings and allows programmers to use additional effects without having to change the signatures, displaying the inferred effects in the IDE heavily improves program understanding. Secondly, the contextual effect system of Effekt makes it very convenient to work with higher-order effect polymorphic functions. Sometimes, we experienced that the effect system is actually too convenient to use: As illustrated in Section 1, changing the type signature of a function influences which effects are handled. These changes might go unnoticed at the call-site and lead to correct, but potentially unexpected behavior. Programmers might assume one particular handler to handle an operation, while that handler is shadowed by a syntactically closer one. To address this, we added warnings as illustrated in Figure 5b that guide the user in writing effectful code.

6.2 Differences to the Calculus

There are some differences between the calculus presented in this paper and its implementation, mostly additional language features. For example, the implementation adds algebraic data types and pattern matching. Importantly, constructors can only take value arguments, no block arguments. The implementation also adds *value type* polymorphism. That is, while Effekt does not include parametric effect polymorphism, parametric polymorphism for value types is supported. There is no interesting interaction, since the universes of blocks, effects, and values are strictly separated.

6.2.1 Builtin Effects. The implementation of Effekt separates user defined effects representing control effects (as presented in Section 2) from builtin effects, such as printing to a console or accessing the network. Like user effects, builtin effects are tracked by the effect system, however, they cannot be handled by users and simply propagate to the top level (that is, the `main` function). In consequence, programs with unhandled builtin effects can be executed, while the type checker prevents programs with unhandled user-defined effects from being evaluated. Since builtin effects do not capture the continuation and cannot be handled by users, they also do not require capability passing. The implementation selectively only performs capability passing for user-defined effects.

6.2.2 Local Mutable State. Algebraic effects have been conceived in the setting of functional programming. Yet, Effekt supports a number of "imperative" features such as iteration with `while` and local mutable state. In presence of control effects, the latter is particularly interesting. Let us assume the example on the left, which is adapted from Brachthäuser et al. [2020a]:

```
try {
  var x = 0
  if (do Choice()) { x = 2 } else { () }
  println(x)
} with Choice { () ⇒
  resume(true); resume(false)
}
```

```
var x = 0
try {
  if (do Choice()) { x = 2 } else { () }
  println(x)
} with Choice { () ⇒
  resume(true); resume(false)
}
```

Here, the `Choice` effect (Section 2) is handled by first resuming with `true` and then resuming again with `false`. In Effekt, mutable variables are treated as effects *local* to the scope they are declared in and `var x = 0; s` can be understood as `state(0) { x ⇒ s }`. Since the `Choice` effect is handled outside of the scope, the example prints `2` and then `0`, which corresponds to separately evaluating the two paths of control flow. Changing the example to the one on the right, which is nested differently, it now prints `2` twice. The change to `2` persists between the two executions since variable `x` is defined *outside* of the handler for `Choice`. The implementation of local mutable state in Effekt follows Brachthäuser et al. [2020a] who in turn build on the work by Kiselyov et al. [2006]: The monad for delimited control is extended by also storing state frames on the stack. On continuation capture, the state is copied and on a call to the continuation it is restored. This way, it is guaranteed that we get the expected interaction between mutable state and delimited control.

6.3 Case Studies

To practically evaluate the language design of Effekt and to analyze the severity of the limitations imposed by treating blocks as second class, we implemented several medium sized case studies. The implemented case studies range over various domains from language implementation techniques (such as lexing, parsing, tree transformations, and pretty printing), over build systems [Mokhov et al. 2018], to natural language DSLs [Maršík and Amblard 2016]. We also implemented many small examples from domains such as pull and push-based streams, probabilistic programming, and automatic differentiation. Here, we report on our experience in developing those case studies, describe encountered limitations and workarounds, and highlight interesting examples related to effect polymorphism and second-class values. We use Koka as a point of comparison, but most of the concepts we illustrate also apply to other languages with row-polymorphic effects and handlers.

6.3.1 Lexer: Pull-Based Streams. Based on the motivating example from Section 2.2, in our case studies we implement a lexer effect with the signature on the left. Like in Koka [Leijen 2017b], effect signatures in our implementation of Effekt can group multiple effect operations.

```
effect Lexer {
  def peek(): Option[Token]
  def next(): Token
}
```

```
try {
  var index = 0;
  def eos(): Boolean = index ≥ in.length
  ...
} with Lexer { ... }
```

Instead of feeding tokens from a given list, we implemented a handler for the `Lexer` effect that analyses a given input string. The handler uses a local mutable reference `index` to store the current position in the string. It also makes extended use of local function definitions, such as `eos` on the right, which close over `index`. In Koka, the type checker would reject the program on the right, since the type annotation on `eos` with the empty effect set does not match the expected type, which in Koka is `local<_h> Boolean`. The Koka language translates mutable variables and references to a synthesized state effect (such as `local<_h>`). Built around row-polymorphism, the details of this encoding and the corresponding synthesized effects leak into the type of user programs [Leijen 2018a]. In contrast, Effekt builds on second-class functions to achieve the same safety. In consequence, there is no such "State" effect that could leak to the user, making it much more natural to work with locally defined effects in general and with local variables in particular. Recalling the typing rule Def (Section 3.2): all effects that are not mentioned on a block must be handled at the definition site of the block. This is safe, since the block `eos` cannot leave the scope of its definition.

6.3.2 Parser. As illustrated in Section 2.4, we can use our extended implementation of lexers to describe parsers and our case studies expand on this idea. Modeling parsers with effects allows us to implement the parsing strategy in the handler. For instance we can describe depth-first strategies which enumerate either only the first, or all possible results. However, the restriction of blocks to be second-class prohibits us from writing handlers for breadth-first strategies [Brachthäuser et al. 2020a]. To implement breadth-first parsers, it would be necessary to store continuations in a queue. However, since continuations are blocks and blocks are second class, we cannot do this in Effekt.

6.3.3 ANF-Transformation. By giving us access to delimited continuations, effect handlers help us to express structured tree transformations. For example, we can implement a transformation of expression trees into A-normal form. Let us assume data types for the abstract syntax of a language that explicitly distinguishes between effectful statements (`Stmt`) and pure expressions (`Expr`).

```
effect Fresh(): String
effect Bind(stmt: Stmt): Expr
```

```
def bindHere { prog: Stmt / Bind } : Stmt / Fresh =
  try { prog() }
  with Bind { (stmt) ⇒
    val id = do Fresh()
    Let(id, stmt, resume(Var(id)))
  }
```

We can use the `Bind` effect operation to locally transform a statement into an expression. The handler `bindHere` handles the `Bind` effect by generating a fresh variable and inserting a binder for the provided statement `stmt`. In Koka, the inferred type of `bindHere` would be

```
forall<e> (prog : () → <fresh,bind|e> int) → <fresh|e> int
```

Here we can see again two important differences: First, Koka uses parametric effect polymorphism and thus quantifies over the effect row `e`. Second, by use of row polymorphism and ML-style unification, the use of `Fresh` in the handler propagates into the type of the argument `prog`. As we discuss in Section 7.1.2, this breaks *effect encapsulation* since it leaks (effect-)implementation details into the signature.

6.3.4 Pretty-Printer. Similar to how parsers can be implemented by backtracking search, we can also use effects and handlers to describe pretty printers. In an extended case study, we implemented a pretty printing library that conceptually resides between the imperative presentation of Oppen [1980] and the functional presentation of Swierstra and Chitil [2009]. It combines multiple different uses of effects, such as effects for dynamically bound variables [Brachthäuser and Leijen 2019], backtracking search, and writing to an output stream. For example, we use effects to abstract over the indentation level and the layouting direction, which are context dependent.

```
type Direction {
  Horizontal();
  Vertical()
}
effect Indent(): Int
effect DefaultIndent(): Int
effect Flow(): Direction
```

```
def direction[R](dir: Direction) { doc: R / Flow }: R =
  try { doc() }
  with Flow { () ⇒ resume(dir) }

def horizontal { p: Unit / Flow }: Unit =
  direction(Horizontal()) { p() }
```

The effect `Flow` represents the current layouting direction and the effect `Indent` represents the context-dependent indentation level. To express the handler `horizontal` on the right, it would be convenient to partially apply the direction function such as:

```
val horizontal = direction(Horizontal()) // ERROR
```

However, in Effekt this is not possible since functions cannot be returned. As a commonly encountered workaround, we have to η-expand the definition of `horizontal`. To express pretty printers, we use two additional effects. An effect to output the rendered document (`Emit`) and an effect for choice-points in the layout (`LayoutChoice`).

```
effect Emit {
  def text(content: String): Unit
  def newline(): Unit
}
```

```
effect LayoutChoice {
  def fail[A](): A
  def choice(): Direction
}
```

The `LayoutChoice` effect is very similar to the non-determinism effect with `Fail` and `Choice` from Section 2. As an example, the pretty-printing combinator `group` [Swierstra and Chitil 2009] uses non-deterministic choice to decide the layouting direction of a document.

```
def group { p: Unit / Flow } =
  direction(choice()) { p() }
```

```
def line() = do Flow() match {
  case Horizontal() ⇒ text(" ")
  case Vertical()   ⇒ newline()
}
```

Using `group` and `line` we can define conditional linebreaks as `group { line() }`, which inserts a linebreak if the remainder does not fit into the available space. Finally, we assemble the handler `pretty` for pretty printed document descriptions from multiple other handlers.

```
def searchLayout[R] { p : R / LayoutChoice }: Option[R]
def writer { p: Unit / Emit }: String
def printer {
  prog: Unit / {  Emit, Indent, DefaultIndent, Flow }
} : Unit / { Emit, LayoutChoice }

effect Pretty = { Emit, Indent, DefaultIndent, Flow, LayoutChoice }
def pretty { doc: Unit / Pretty }: Option[String] =
  searchLayout { writer { printer { doc() } } }
```

Again, in Koka the above would not typecheck. The signature of printer would be polymorphic in an effect row `e`.

```
forall<e> (() → <emit,indent,defaultIndent,flow|e> int) → <emit,layoutChoice|e> int
```

Since `doc` can use `LayoutChoice` in addition to effects that are explicitly mentioned in the type of `prog`, at the callsite of `printer` the row variable `e` would be instantiated to `LayoutChoice`. In consequence, the inferred type of the call to `printer` is `<emit, layoutChoice, layoutChoice> ()`. It contains *two* copies of `LayoutChoice` in its row. Only one copy is handled by `searchLayout` and the overall type of `pretty` thus still mentions `LayoutChoice`! We can resolve this problem by adding `LayoutChoice` to the type of `prog` in the signature of `printer`. But this is a change to the definition of `printer` that is motivated by a particular use of `printer`, and the signature of `printer` would then say that its implementation could handle the `LayoutChoice` effect, which it does not do. While it is sometimes possible to resolve problems like this by changing type signatures of involved functions or by advanced use of term level adjustments [Convent et al. 2020; Leijen 2018a], lexically scoped effects and contextual effect polymorphism in Effekt avoid this class of problems altogether.

6.3.5 Automatic Differentiation. The restriction to second-class blocks implies that blocks can neither be returned from functions, nor can they be stored in data structures. One common use of

the former is curried application. A common workaround is to η-expand the definition. In order to allow storing blocks in data structures, one can manually perform defunctionalization. While in some cases this is a viable approach, some application domains inherently rely on function spaces.

One example of such a domain is automatic differentiation (AD). Wang et al. [2019] propose to use delimited continuations to implement reverse-mode AD. While the presented techniques readily carry over to effect handlers (and can be applied in Effekt), in this domain it would be natural to talk about functions as a first-class concept. For example, one could try to implement a handler `grad` that takes a function `Num ⇒ Num / AD`, which uses the `AD` effect, to its derivative.

```
def grad { prog: Num ⇒ Num / AD } : Double ⇒ Double
```

The returned function would close over `prog` which could have arbitrary other effects. In Effekt, this is not possible and we have to define `grad` with the following signature:

```
def grad(in: Double) { prog: Num ⇒ Num / AD } : Double
```

Another domain that suffers from a similar restriction is probabilistic programming. While effect handlers have been shown to be suitable to modularize the description of models and inference algorithms [Bingham et al. 2019; Moore and Gorinova 2018; Phan et al. 2019], models are typically represented by functions and it is necessary to treat them as first class. Manual defunctionalization of models is possible in some cases but requires work and the result is less natural.

7 DISCUSSION AND RELATED WORK

In this paper, we introduced a language with effect handlers, based on a new reading of effect types as contextual requirements. In this section, we compare Effekt with other implementations of effect handlers and discuss prior work related to the contextual reading (summarized in Figure 6).

7.1 Effect Handlers

Section 2 illustrated that, despite the different reading of effect types, programming in Effekt with first-order functions is not much different from programming in other languages with effect handlers. However, for higher-order functions we can observe significant differences. Let us assume the following implementation for our motivating example from Section 1.

```
def eachLine(file: File) { f: String ⇒ Unit / {} }: Unit / { FileIO, Console } =
  try {
    while (hasNext(file)) { f(nextLine(file)) }
  } with Exception { () ⇒ println("error reading file") }
```

The implementation of `eachLine` handles exceptions that arise from the interaction with files via `hasNext` and `nextLine` by printing an error message.

7.1.1 Accidental Capture. In our motivating example the block passed to `eachLine` used exceptions:

```
eachLine { (line) ⇒ if (line == "") { do Exception() } else { ... } }
```

What happens if the file indeed does contain an empty line? In many languages with support for exception handling (like Java, JavaScript, OCaml, and others) the exception will be caught by the dynamically closest handler. This example would thus print `"error reading file"`. For the user of `eachLine` this might come as a surprise, as she would have expected it to print `"empty line"`. This behavior exposes an implementation detail of `eachLine`, which is clearly undesirable. We call this behavior *accidental capture*, since the type of `eachLine` does not mention `Exception` and `eachLine` should not be able to intercept exceptions.

	Effect Safe	Capture-free	Effect Handlers	Encapsulation without Lifts	Contextual Effect Polym.
Effect handlers without static checks					
Plotkin and Pretnar [2013]	✗	✗	✓	–	–
Dolan et al. [2017]	✗	✗	✓	–	–
Brachthäuser and Schuster [2017]	✗	✓	✓	–	–
Effect handlers with static checks					
Bauer and Pretnar [2013]	✓	✓	✓	–	–
Leijen [2017b]	✓	✓	✓	✗	✗
Lindley et al. [2017]	✓	✓	✓	✗	✗
Hillerström and Lindley [2016]	✓	✓	✓	✗	✗
Lexical effect handlers with static checks					
Zhang and Myers [2019]	✓	✓	✓	✓	✗
Brachthäuser et al. [2020a]	✓	✓	✓	✓	✗
Biernacki et al. [2020]	✓	✓	✓	✓	✗
Second-class values					
Osvald et al. [2016]	✓	–	✗	✓	✓
Zhang et al. [2016]	✓	✓	✗	✓	✓
Effekt	✓	✓	✓	✓	✓

Fig. 6. Summary of related work.

Zhang et al. [2016] propose a solution to avoid accidental capture in the context of exceptions in Java-like languages. They distinguish between code that is aware of a certain exception and oblivious code. Only code that is aware of an exception can handle them, exceptions are *tunneled* otherwise. They also translate methods that throw exceptions to receive and pass along an additional parameter that will be a label at runtime, similar to labels in System Ξ. Furthermore, they also distinguish first-class (precise) and second-class (imprecise) functions and prevent the latter ones from being returned. Their calculus features parametric effect (*i.e.*, *label*) polymorphism, which is inferred and hidden from the user. To achieve this, they use intraprocedural program analysis to solve constraints arising from the use of exceptions and handlers and generalize at procedure boundaries. Their type system is strictly more general, as it allows for first-class functions when they are safe to use, but arguably also more complicated. While they are concerned solely with exceptions, in this work we are concerned with the more general concept of effect handlers, which requires a sound treatment of resumptions.

7.1.2 Dynamically Scoped Effects. The problem of accidental capture also arises in languages with effect handlers that have a dynamic semantics based on searching the stack for a matching handler [Dolan et al. 2014; Leijen 2017b; Lindley et al. 2017; Plotkin and Pretnar 2013]. These languages can be grouped according to whether they statically check effects or not.

Languages without effect systems. Languages without a static effect system like Eff [Plotkin and Pretnar 2013] or MultiCore OCaml [Dolan et al. 2014] do not guarantee effect safety and accidental capture happens silently. By absence of an effect system, these languages support effect polymorphic reuse, trivially.

Effect systems based on rows. Languages with effect-systems based on row polymorphism like Koka [Leijen 2017b], Frank [Lindley et al. 2017], or Links [Hillerström and Lindley 2016] guarantee effect safety and support effect polymorphism in the form of parametric effect polymorphism.

While these languages still exhibit accidental capture, this behavior is less surprising because it is reflected in the types. In Koka, for example, the inferred signature of `eachLine` would be:

```
eachLine : (file, string → <exception|e> ()): <fileio,console|e> ()
```

This type clearly states that `eachLine` can intercept exceptions thrown by its function argument. However, there is a flip side of the coin: the fact that `eachLine` uses exceptions in its implementation leaks into the type of its parameter `f`. This correctly reflects the runtime semantics: exceptions raised by `f` are handled by `eachLine` resulting in `"error reading file"`. To *encapsulate* this implementation detail, some languages require the user to manually use term-level lifting constructs [Biernacki et al. 2017; Convent et al. 2020; Leijen 2018b]. For example, in Koka the call to the argument function `f` would need to be enclosed in a lift to express that any exceptions in `f` should not be handled by the next enclosing handler `inject<exception>({ f(nextLine(file)) })`. These lifting constructs come with their own complications [Convent et al. 2020] and our own experience working in these languages confirms that they pose a major usability issue, even in medium-sized programs. The need for them is a consequence of the operational semantics based on dynamic handler search in these languages.

7.1.3 Lexically Scoped Effects. More recent languages with effect handlers implement *lexically scoped effects*. In those languages, effect operations are not handled by their dynamically closest handler, but instead a *lexical* relation between handlers and effect operations is established. These languages do not suffer from accidental capture. Again, we can organize the languages according to the presence of a static effect system.

Languages without effect systems. Like System Ξ, implementations based on explicit [Brachthäuser et al. 2018] or implicit [Brachthäuser and Schuster 2017] capability passing avoid the problem of accidental capture by closing over the lexically scoped capability. Bauer and Pretnar [2015] explicitly pass effect instances and avoid the problem of accidental capture. Without a static effect system, those languages do not guarantee effect safety.

Languages without effect polymorphism. Bauer and Pretnar [2013] present an effect system for a language with effect handlers. They assume a statically fixed set of *effect instances*, which are term-level constants. Instances are in spirit similar to capabilities and allow programmers to disambiguate operations for the same effect, corresponding to different handlers. In contrast to our capabilities, instance expressions are first-class and their types have to statically approximate the (set of) effect instances the expression could stand for. While in System Ξ, capabilities are created and bound at handlers, handlers in the work of Bauer and Pretnar [2013] can be parametrized by the effect instance. In some cases those instances are not known statically, and thus the handler cannot remove the instance from the effect type of the handled expression. The most important difference to the present work, is that the language that they present does not feature effect polymorphism. Instead, in first-order programs code reuse at different effect types is achieved with subtyping. They, however, do not discuss how to achieve code reuse of higher-order functions.

Schuster et al. [2020] present a technique for compiling effect handlers in capability-passing style. Just like our language System Ξ, their language λ_{Cap} is in explicit capability-passing style. They explore compiler optimizations for languages with effect handlers, while we explore effect systems from a user perspective. While System Ξ does not have an effect system and relies on blocks being second-class to ensure effect safety, their language indexes effectful computations with a list of answer types to ensure effect safety. They do not feature effect polymorphism and assume that all programs have been monomorphized. They use the statically known list of answer types to translate their programs to iterated continuation-passing style [Schuster and Brachthäuser 2018],

while we use a library for multi-prompt delimited continuations [Dybvig et al. 2007]. It would be interesting to connect the two approaches by inferring the list-structured effect types for System Ξ.

Languages with parametric effect polymorphism. Languages with lexically scoped handlers are equipped with effect systems [Biernacki et al. 2020; Brachthäuser et al. 2020a; Zhang and Myers 2019] and track a set of effect instances on the type level. By establishing a lexical binding, lifting annotations are not required while guaranteeing effect parametric reasoning [Zhang and Myers 2019]. However, the effect systems of these languages are typically quite involved and often require limited forms of dependent types. Furthermore, they only support *parametric* effect polymorphism.

Xie et al. [2020] provide a connection between dynamically scoped effects and lexcically scoped effects as a translation from a language with the former to a language with the latter. They prove that on a subset of the former, which only uses *scoped resumptions*, the semantics coincide. Their source language features effect polymorphism based on effect rows [Leijen 2014] while our source language is based on contextual effect polymorphism. Guided by the effect types, they translate programs to explicitly pass *evidence vectors*, very similar to how we translate programs to explicitly pass individual *capabilities*.

In summary, in comparison to other languages with effect handlers, Effekt uniquely guarantees effect safety, avoids accidental capture, and establishes encapsulation without lifts with a very lightweight effect system based on contextual effect polymorphism.

7.2 Second-Class Values

Our treatment of functions as second-class values is inspired by Osvald et al. [2016]. They use second class functions to establish a type-based escape analysis and present case studies for exceptions, memory regions, and well-scopedness in program generation. In this paper, we generalize the calculus of Osvald et al. [2016] to effect handlers, and present a language design around this insight.

The work by Osvald et al. [2016] and Zhang et al. [2016] suggests that it is viable to add first-class functions to Effekt. However, we purposefully refrained from doing so to focus on studying the combination of effect handlers and second-class functions (blocks). While a large class of programs using effect handlers is still expressible in Effekt despite this restriction, we want to single out three representative examples from the literature that are not.

Functional State. Effect handlers can encode state [Pretnar 2015] by interpreting the `Get` and `Put` operations into an answer type that is a function. In Effekt, this is not allowed since handlers cannot return functions. The idiomatic way in Effekt to write a handler for `Get` and `Put` is to use a local reference, which is safe and has the correct backtracking behavior as described in Section 6.2.2

Scheduler. Effect handlers can express structured concurrency with user defined schedulers [Dolan et al. 2015]. In Effekt, this is not possible since it would require to store the continuation in a data structure (such as a queue). However, in Effekt continuations are blocks and can neither be returned, nor stored in data structures. In our implementation of Effekt we resort to locally unsafe primitives.

Scoped Effects. Effect operations that take function parameters are called *higher-order operations* or *scoped* [Wu et al. 2014]. To establish effect safety, effect operations in Effekt cannot take block arguments. Allowing effect operations to take block arguments would break effect safety, since this way blocks could close over capabilities that in turn would leave their defining scope [Brachthäuser et al. 2020a]. This way, examples like the async effect by Leijen [2017a] are ruled out in Effekt:

```
effect async { fun await( initiate : (result<a> → io ()) → io ()) : result<a> }
```

However, it is possible to define an API in Effekt that is based on promises [Dolan et al. 2017].

7.3 Coeffects

The general idea of expressing requirements on the context in the type is not new and can in similar form be found in coeffect systems [Petricek et al. 2014]. Petricek et al. describe a calculus that can be instantiated to express (amongst others) implicit variables. We also interpret effects as a requirement on the calling context. In consequence, some details of the effect system of Effekt align with their coeffect system for implicits. In particular, effects in Def (Figure 3.2) can also be handled either at the call-site or the definition-site. Petricek et al. [2014] refer to those requirements as *latent* or *immediate*, correspondingly. While the effect system of Effekt has some similarities to coeffects, we also allow for control effects. In consequence, while implicit variables (once resolved) are treated as first class, the type-system of Effekt needs to assert that capabilities cannot escape the region of their handler. Effekt can thus be seen as a combination of a coeffects, second-class values, and delimited control. In this paper, we present a practical language design employing the contextual perspective. In the future, we hope to formally establish the relation to coeffects.

7.4 Lightweight Polymorphic Effects

Rytz et al. [2012] introduces separate function types for effect monomorphic and effect polymorphic functions. Applying the latter, the effects of the argument function still have to be handled at the call-site. This is similar to contextual effect polymorphism and all functions (*i.e.*, blocks) in Effekt are effect polymorphic in the sense of Rytz et al. [2012]. However, in the calculus of Rytz et al. [2012], functions are either fully polymorphic or monomorphic. It is thus not clear how to express handler functions like $\texttt{always}_{42}$ (Section 2) that mention some effects on their argument functions, but are polymorphic in the rest.

Summary. The design of Effekt aims at exploring a sweet spot between simplicity and expressiveness. In particular, the Effekt language: guarantees effect safety – all user defined control effects are eventually handled; avoids accidental capture – effects not mentioned in a signature cannot be observed; supports effect handlers – advanced control-flow constructs can be expressed as libraries; guarantees effect encapsulation – implementation effects are not leaked into the signature; supports contextual effect polymorphism – higher order functions can be used effect polymorphically without having to quantify effect variables.

8 CONCLUSION

In this paper, we have taken a novel view on effect systems for languages with effect handlers. Traditionally, effect types express which *side effects* a computation might have, besides computing the resulting value. Instead, we have assumed a different interpretation where effect types express which *capabilities* a computation requires from its context. This different perspective opens up a novel design space, offering different trade-offs. On the negative side, we lose purity guarantees since contextual purity does not imply purity. The restriction to blocks instead of first-class functions also entails some expressiveness limitations. On the positive side, the language design becomes very simple. Parametric effect polymorphism is not needed anymore (and replaced by lightweight contextual effect polymorphism) and we can guarantee effect safety and absence of accidental capture. We believe that we have identified an interesting sweet spot of effect handler design that can contribute to a more wide-spread adoption of effect handlers in programming languages.

ACKNOWLEDGMENTS

We are very grateful for the valuable feedback by the anonymous reviewers. This work was supported by DFG project 282458149.

REFERENCES

Henk P. Barendregt. 1992. Lambda Calculi with Types. In *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures*. Oxford University Press, New York, NY, USA, 117–309.

Andrej Bauer and Matija Pretnar. 2013. An effect system for algebraic effects and handlers. In *International Conference on Algebra and Coalgebra in Computer Science*. Springer, Berlin, Heidelberg, 1–16.

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123.

Nick Benton, Chung-Kil Hur, Andrew J Kennedy, and Conor McBride. 2012. Strongly typed term representations in Coq. *Journal of automated reasoning* 49, 2 (2012), 141–159.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 2, POPL, Article 8 (Dec. 2017), 30 pages.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Abstracting Algebraic Effects. *Proc. ACM Program. Lang.* 3, POPL, Article 6 (Jan. 2019), 28 pages.

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by Day, Labels by Night: Effect Instances via Lexically Scoped Handlers. In *Proceedings of the Symposium on Principles of Programming Languages (to appear)*. ACM, New York, NY, USA.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20, 1 (Jan. 2019), 973–978.

Jonathan Immanuel Brachthäuser and Philipp Schuster. 2017. Effekt: Extensible Algebraic Effects in Scala (Short Paper). In *Proceedings of the International Symposium on Scala* (Vancouver, BC, Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3136000.3136007

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 111 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276481

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020a. Effekt: Capability-Passing Style for Type- and Effect-safe, Extensible Effect Handlers in Scala. *Journal of Functional Programming* (2020). https://doi.org/10.1017/S0956796820000027

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020b. *Effekt: Lightweight Effect Polymorphism for Handlers*. Extended Technical Report. University of Tübingen, Germany. http://ps.informatik.uni-tuebingen.de/publications/brachthaeuser20effekt.pdf.

Jonathan Immanuel Brachthäuser and Daan Leijen. 2019. *Programming with Implicit Values, Functions, and Control*. Technical Report MSR-TR-2019-7. Microsoft Research.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

Lukas Convent, Sam Lindley, Conor McBride, and Craig McLaughlin. 2020. Doo Bee Doo Bee Doo. *Journal of Functional Programming* 30 (2020), e9. https://doi.org/10.1017/S0956796820000039

Olivier Danvy and Andrzej Filinski. 1989. A functional abstraction of typed contexts. *DIKU Rapport 89/12, DIKU, University of Copenhagen* (1989).

Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, KC Sivaramakrishnan, and Leo White. 2017. Concurrent system programming with effect handlers. In *Proceedings of the Symposium on Trends in Functional Programming*. Springer LNCS 10788.

Stephen Dolan, Leo White, and Anil Madhavapeddy. 2014. Multicore OCaml. In *OCaml Workshop*.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective concurrency through algebraic effects. In *OCaml Workshop*.

R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. 2007. A monadic framework for delimited continuations. *Journal of Functional Programming* 17, 6 (2007), 687–730.

Richard A. Eisenberg, Joachim Breitner, and Simon Peyton Jones. 2018. Type Variables in Patterns. In *Proceedings of the Haskell Symposium* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3242744.3242753

Matthias Felleisen. 1988. The Theory and Practice of First-class Prompts. In *Proceedings of the Symposium on Principles of Programming Languages* (San Diego, California, USA). ACM, New York, NY, USA, 180–190.

Carl A. Gunter, Didier Rémy, and Jon G. Riecke. 1995. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA). ACM, New York, NY, USA, 12–23.

Daniel Hillerström and Sam Lindley. 2016. Liberating Effects with Rows and Handlers. In *Proceedings of the Workshop on Type-Driven Development* (Nara, Japan). ACM, New York, NY, USA.

Daniel Hillerström, Sam Lindley, Bob Atkey, and KC Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *Formal Structures for Computation and Deduction (LIPIcs, Vol. 84)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the International Conference on Functional Programming* (Boston, Massachusetts, USA). ACM, New York, NY, USA, 145–158.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. 2006. Delimited Dynamic Binding. In *Proceedings of the International Conference on Functional Programming* (Portland, Oregon, USA). ACM, New York, NY, USA, 26–37.

Daan Leijen. 2005. Extensible records with scoped labels. In *Proceedings of the Symposium on Trends in Functional Programming*. 297–312.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

Daan Leijen. 2016. *Algebraic Effects for Functional Programming.* Technical Report. MSR-TR-2016-29. Microsoft Research technical report.

Daan Leijen. 2017a. Structured Asynchrony with Algebraic Effects. In *Proceedings of the Workshop on Type-Driven Development* (Oxford, UK). ACM, New York, NY, USA, 16–29.

Daan Leijen. 2017b. Type directed compilation of row-typed algebraic effects. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 486–499.

Daan Leijen. 2018a. *Algebraic Effect Handlers with Resources and Deep Finalization.* Technical Report MSR-TR-2018-10. Microsoft Research. 35 pages.

Daan Leijen. 2018b. First Class Dynamic Effect Handlers: Or, Polymorphic Heaps with Dynamic Effect Handlers. In *Proceedings of the Workshop on Type-Driven Development* (St. Louis, Missouri, USA). ACM, New York, NY, USA, 51–64.

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210.

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the Symposium on Principles of Programming Languages* (Paris, France). ACM, New York, NY, USA, 500–514.

Jirka Maršík and Maxime Amblard. 2016. Introducing a Calculus of Effects and Handlers for Natural Language Semantics. In *International Conference on Formal Grammar*. Springer LNCS 9804, 257–272.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (July 2018), 29 pages. https://doi.org/10.1145/3236774

Dave Moore and Maria Ivanova Gorinova. 2018. Effect Handling for Composable Program Transformations in Edward2. In *International Conference on Probabilistic Programming (PROBPROG)*.

Dereck C Oppen. 1980. Prettyprinting. *Transactions on Programming Languages and Systems* 2, 4 (1980), 465–483.

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I González Alayón, and Tiark Rompf. 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*. ACM, New York, NY, USA, 234–251.

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the International Conference on Functional Programming* (Gothenburg, Sweden). ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable effects for flexible and accelerated probabilistic programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013).

Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *Proceedings of the European Conference on Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–282.

Philipp Schuster and Jonathan Immanuel Brachthäuser. 2018. Typing, Representing, and Abstracting Control. In *Proceedings of the Workshop on Type-Driven Development* (St. Louis, Missouri, USA). ACM, New York, NY, USA, 14–24. https://doi.org/10.1145/3240719.3241788

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. 4, ICFP, Article 93 (Aug. 2020), 28 pages. https://doi.org/10.1145/3408975

Dorai Sitaram. 1993. Handling Control. In *Proceedings of the Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA). ACM, New York, NY, USA, 147–155.

S Doaitse Swierstra and Olaf Chitil. 2009. Linear, bounded, functional pretty-printing. *Journal of Functional Programming* 19, 01 (2009), 1–16.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Essertel, and Tiark Rompf. 2019. Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator. *Proc. ACM Program. Lang.* 3, ICFP, Article 96 (July 2019), 31 pages.

Andrew K. Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Inf. Comput.* 115, 1 (Nov. 1994), 38–94.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the Haskell Symposium* (Gothenburg, Sweden) *(Haskell '14)*. ACM, New York, NY, USA, 1–12.

Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect Handlers, Evidently. 4, ICFP, Article 99 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408981

Jeremy Yallop. 2017. Staged Generic Programming. *Proc. ACM Program. Lang.* 1, ICFP, Article 29 (Aug. 2017), 29 pages.

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe Effect Handlers via Tunneling. *Proc. ACM Program. Lang.* 3, POPL, Article 5 (Jan. 2019), 29 pages.

Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. 2016. Accepting Blame for Safe Tunneled Exceptions. In *Proceedings of the Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA). ACM, New York, NY, USA, 281–295.